Ein Echtzeitparallelrechner zur Rezentralisierung von Steuergeräten im Automobil

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt der

Fakultät für Mathematik/Informatik und Maschinenbau

der Technischen Universität Clausthal

am 23.10.2012

von
Stefan Aust
geboren am 30.04.1974 in Eisenhüttenstadt

Gutachter:
Prof. Dr.-Ing. Dr. rer. nat. habil. Harald Richter
Prof. Dr. rer. nat. Christian Siemers
Prof. Dr.-Ing. Christian Bohn

Datum der Verteidigung: 21.08.2013

Kurzzusammenfassung

Die Funktionalität und das Verhalten moderner Fahrzeuge werden zunehmend durch Elektronik und durch Software bestimmt. Zahlreiche Assistenzsysteme unterstützen den Fahrer durch aktive Sicherheit und Komfort, intelligente Diagnosesysteme erleichtern die Fehlersuche und neue Kommunikationssysteme vermeiden Verkehrsstau. Infolgedessen steigen jedoch der Bedarf an Software ebenso wie die Zahl der elektronischen Steuergeräte rasant an.

Die vorliegende Arbeit geht der Frage nach, welche Probleme durch die gegenwärtige Entwicklung in der E/E-Architektur moderner Fahrzeuge entstehen, und zeichnet eine mögliche Lösung dieser Probleme durch die Rezentralisierung von Steuergeräten in einem Echtzeitparallelrechner auf. Dazu werden die besonderen Eigenschaften von Echtzeitrechnern und von Parallelrechnern, welche sich in der Historie unabhängig voneinander entwickelt haben, in der neuen Rechnerklasse der Echtzeitparallelrechner vereint. In diesem Rahmen wurde das Konzept des Space-Sharings entwickelt, welches zusammen mit dem ebenfalls neu vorgestellten FPGA-basierten Software-First-Design die Grundlage für den Entwurf des Echtzeitparallelrechners ConPar darstellt.

Eine ganz wesentliche Bedeutung kommt dem Verbindungsnetzwerk zu, welches im Echtzeitparallelrechner die Interprozessorkommunikation realisiert und sowohl skalierbar als auch echtzeitfähig sein muss. Hierfür wurde eine Analyse verschiedener etablierter Topologien statischer und dynamischer Netze durchgeführt, auf deren Basis zwei mögliche Lösungen erarbeitet wurden.

Da ConPar für den Einsatz im Automobil konzipiert wurde, findet auch das Thema der Energieeffizienz besondere Berücksichtigung. Auf der Basis von Space-Sharing werden mehrere Methoden vorgestellt, um die Verlustleistung des Echtzeitparallelrechners effektiv zu minimieren. Darüberhinaus wird die Integration von ConPar in die bestehende Toolkette von AUTOSAR, dem aktuellen Standard in der Softwareentwicklung automobiler Steuergeräte, betrachtet.

Danksagung

Die vorliegende Dissertation entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter in der Abteilung „Technische Informatik und Rechnersysteme“ am Institut für Informatik der Technischen Universität Clausthal.

Mein erster Dank geht natürlich an meinen Betreuer, *Herrn Prof. Dr. Harald Richter*. Seine Idee zum Entwurf eines Echtzeitparallelrechners zur Rezentralisierung von Steuergeräten hat sich als ein sehr spannendes und zukunftsträchtiges Thema erwiesen, dessen interessante Aspekte und Fragestellungen mich über die gesamte Zeit immer wieder neu motiviert haben. Darüber hinaus danke ich ihm auch für seine Unterstützung bei der Präsentation meiner Forschungsergebnisse auf zahlreichen nationalen und internationalen Konferenzen.

Mein weiterer Dank gilt *Herrn Prof. Dr. Christian Siemers* und *Herrn Prof. Dr. Christian Bohn* für ihre Tätigkeit als Nebengutachter.

In diesem Zusammenhang möchte ich *Herrn Prof. Dr. Christian Bohn* und *Herrn Prof. Dr. Oliver Zirn* dafür danken, dass sie meine Zulassung zum Promotionsstudium durch ihre Gutachten über mein Vorhaben unterstützt haben.

Mein ganz besonderer Dank gilt *Herrn Prof. Dr. Christian Siemers* und *Herrn Dr. Christian Vetter* für ihr großes Interesse an meiner Arbeit, ihre stete Ansprechbarkeit und ihre hilfreichen Hinweise und Impulse, welche sie mir in unseren gemeinsamen Gesprächen mitgegeben haben.

Nicht vergessen möchte ich auch *Prof. Dr. Erhard Stein*, der mich durch seine Empfehlung erst auf den Weg zur Promotion gebracht hat. Vielen Dank hierfür!

Und nicht zuletzt meinen herzlichsten Dank an meine Ehefrau, *Frauke Aust*, die mein Promotionsvorhaben von Anfang bis Ende unterstützt hat.

Inhalt

Abkürzungen

ABS	-	Antiblockiersystem
API	-	Application Programming Interface
AUTOSAR	-	AUTomotive Open System ARchitecture
BCET	-	Best Case Execution Time
CAN	-	Controller Area Network
DMA	-	Direct Memory Access
ECU	-	Electronic Control Unit
EDK	-	Embedded Development Kit
ESP	-	Elektronisches Stabilitätsprogramm
FFT	-	Fast Fourier Transformation
FPGA	-	Field Programmable Array
FPU	-	Floating Point Unit
FSL	-	Fast Simplex Link
GALS	-	Globally Asynchronous, Locally Synchronous
HW	-	Hardware
IP	-	Intellectual Property
ISE	-	Integrated Software Environment
LIN	-	Local Interconnect Network
LMB	-	Local Memory Bus
LSB	-	Least Significant Bit

LUT - Lookup Tabelle

MB - MicroBlaze

MIMD - Multiple Instruction, Multiple Data

MOST - Media Oriented Systems Transport

MPSoC - Multi-Processor System-on-Chip

MSB - Most Significant Bit

NoC - Network-on-Chip

PLB - Processor Local Bus

PSM - Prozessor-Speicher-Modul

RTE - Run-Time Environment

SDK - Software Development Kit

SoC - System-on-Chip

SPOF - Single Point of Failure

SW - Software

UML - Unified Modeling Language

VFB - Virtual Functional Bus

VLSI - Very Large Scale Integration

WCET - Worst Case Execution Time

XPS - Xilinx Platform Studio

Kapitel 1 - Einleitung

Die technische Entwicklung des Automobils. Die Erfindung des Automobils gehört mit Sicherheit zu den größten technischen Erfolgen des 19. Jahrhunderts. Und so dynamisch wie sich das Auto für uns heute als Möglichkeit der individuellen Fortbewegung präsentiert, so dynamisch ist auch dessen stete Weiterentwicklung. Trotz seiner mehr als 100-jährigen Geschichte ist das Automobil aber selbst in der heutigen Form noch weit davon entfernt, als technisch vollendete Lösung für den Individualverkehr zu gelten. Die aktuell geführten Diskussionen um alternative Antriebsmethoden oder „mitdenkende“ Autos sind ein sicherer Beweis für den mittelfristig zu erwartenden technischen Fortschritt auf diesem Gebiet.

Mobile Hochleistungsrechner. Die Erhöhung von Sicherheit und Komfort durch fahrerunterstützende Systeme erfordert zunehmend den Einsatz von Software und damit auch mehr Rechenleistung im Fahrzeug. Die vorliegende Arbeit beschäftigt sich aus diesem Grund in erster Linie mit der Frage, wie eine geeignete Rechnerarchitektur beschaffen sein muss, damit die erforderliche Rechenleistung im Fahrzeug zur Verfügung gestellt werden kann. Hierbei spielen vor allem die notwendigen Fragen der Verarbeitung von großen Mengen an Echtzeitdaten, dem Task-Scheduling, der Interprozessorkommunikation und dem Energieverbrauch eine bedeutende Rolle. Zusätzlich dazu werden einige weitere wichtige Randbedingungen wie die Einbindung der vorgestellten Ideen und Konzepte in bestehende Entwicklungsprozesse und -standards der Automobilelektronik betrachtet.

1.1 Stand der Automobilelektronik

1.1.1 Historische Entwicklung

Einführung der Automobilelektronik. Der große Durchbruch begann für die Automobilelektronik in den frühen 1980er Jahren. So wurde 1978 von der Firma Bosch das elektronische ABS als eines der ersten elektronischen Systeme im Bereich der aktiven Fahrsicherheit vorgestellt. Auch bei der Motorsteuerung wurde die bis dato rein mechanisch arbeitende Unterbrecherzündung durch eine Transistorzündung ersetzt. In den folgenden Jahren kamen weitere Funktionen hinzu und so wurde aus einer einfachen elektronischen Zündung ein komplettes Motormanagement mit kennfeldgestützter Einspritzung und Abgaskontrolle.

Kommunikation zwischen Steuergeräten. Bis Ende der 1980er Jahre bestanden elektronische Systeme im Auto i.d.R. aus einzelnen, nicht vernetzten Steuergeräten. Dieser Zustand änderte sich durch den Einzug neuer komplexer Funktionen, die als verteilte Anwendung auf mehreren Steuergeräten realisiert sind. Als Beispiel sei hier das Elektronische Stabilitätsprogramm (ESP) genannt, bei dem Bremse, Motor und Lenkung eng miteinander kooperieren müssen, um die Längs- und die Querdynamik des Fahrzeugs kontrollieren zu können. Der stark steigende Kommunikationsbedarf zwischen den Steuergeräten führte zur Einführung von CAN als ein Bussystem, welches die bis dahin diskret verdrahteten Signale auf einem einzigen gemeinsamen Kommunikationsmedium bündelte. Nachdem die Kommunikationsleistung von CAN noch bis in die späten 1990er Jahre ausreichte, hat der massive Einsatz von Elektronik im Auto letztlich dazu geführt, dass heute noch einige weitere anwendungsspezifische Bussysteme im Fahrzeug koexistieren. So stellen aufwendige Regelungssysteme, wie z.B. elektronisch gesteuerte Fahrwerke, neue Anforderungen an die Datenübertragung. Aus diesem Grund entstand im Jahr 2000 das FlexRay-Konsortium, welches seine Arbeit im Jahr 2010 mit der Spezifikation der Version 3.0.1 beendet hat[1].

1.1.2 Gegenwärtige Probleme

Hohe funktionale Komplexität. Gegenwärtig erreicht die Elektronik im Automobil eine Komplexität, die neue Schwierigkeiten u.a. bei der Gestaltung der E/E-Architektur, der Systemintegration und der Qualitätssicherung mit sich bringt. Vor allem der stark wachsende Bereich der Fahrerassistenzsysteme benötigt zu dessen funktionaler Realisierung eine Vielzahl an Sensorik und Aktorik im Fahrzeug. Das über das gesamte Fahrzeug verteilte System aus Steuergeräten, welches über viele Jahre historisch gewachsen ist, verstärkt das Problem der Komplexität zusätzlich.

Verarbeitung von Echtzeitdaten. Die steigende Zahl an Sensorik im Fahrzeug erzeugt ein ebenfalls steigendes Volumen an Daten, welche von den Steuergeräten in Echtzeit verarbeitet werden müssen. Neben einer hohen Rechenleistung erfordert die wachsende Komplexität automobiler Software neue Methoden bei Multitasking in Echtzeitsystemen wie auch bei der Kommunikation der datenverarbeitenden Systeme im Fahrzeug. Ein möglicher Ansatz für neuartige Echtzeitrechensysteme ist deshalb im Kapitel 4 - Space-Sharing, S. 46 ff. beschrieben.

[1] Webseite des FlexRay-Konsortiums: http://www.flexray.com/ (Stand: 05.04.2012)

Komfortfunktionen. Ein großer Teil der heutigen Automobilelektronik dient dem Komfort der Fahrzeuginsassen. So werden ursprünglich mechanisch betätigte Funktionen heute durch Elektronik kontrolliert, wie der elektrisch verstellbare Außenspiegel, der elektrische Fensterheber oder die Sitzverstellung mit Memory-Funktion. Zusätzlich kamen eine Reihe neuer Funktionen hinzu, wie die elektrische Sitzheizung oder der Sitz mit integrierter Massagefunktion.

Aktive Fahrsicherheit. Neben dem Komfort gewinnt auch die Fahrsicherheit immer mehr an Bedeutung. Ein Beispiel hierfür ist die jüngste Einführung einer intelligenten Steuerung der Scheinwerfer[2], welche für jede Fahrsituation die geeignete Beleuchtung wählt und z.B. bei entgegenkommenden Fahrzeugen automatisch abblendet. Es ist zwar unbestritten komfortabel, das Auf- und Abblenden der Scheinwerfer vollständig der Automatik zu überlassen, gleichzeitig wird aber auch der Gegenverkehr automatisch vor Blendlicht geschützt, was die Verkehrssicherheit deutlich erhöht. Die Verkehrssicherheit findet sich auch in dem heute bedeutendsten Bereich der Automobilelektronik wieder: dem Bereich der aktiven-Sicherheitssysteme und Fahrerassistenten. Während auf dem Gebiet der passiven Fahrsicherheit mit dem heutigen Stand kaum noch grundlegende Innovationen zu

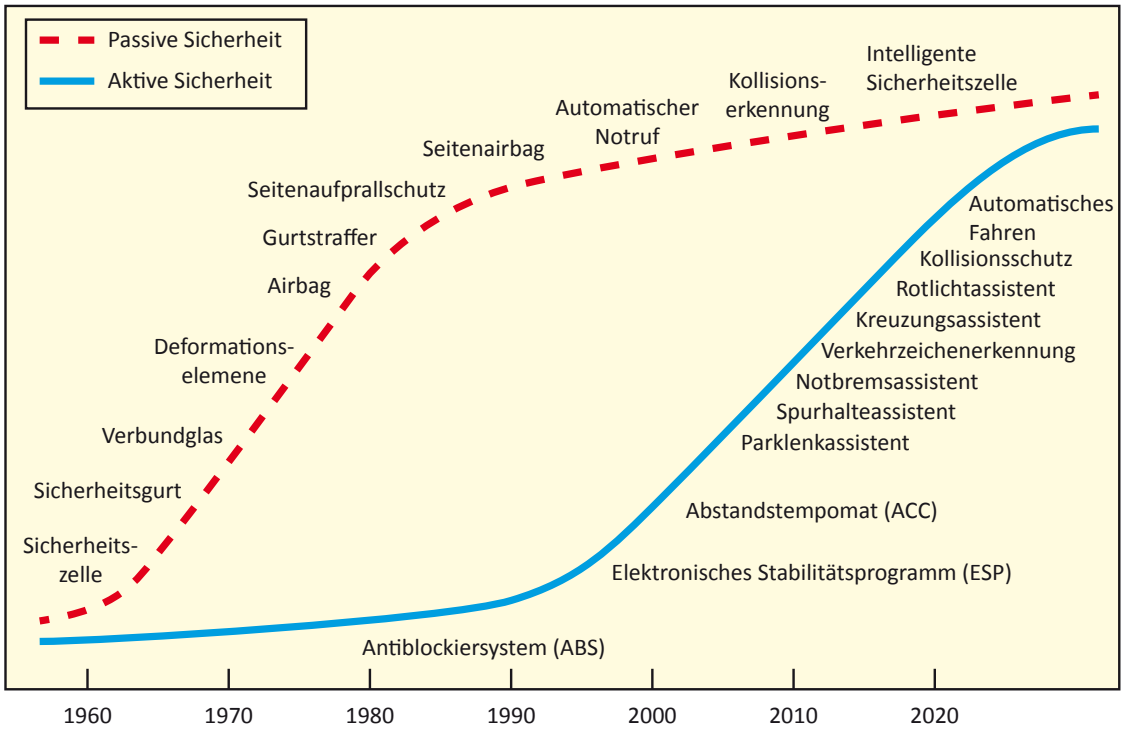


Abbildung 1.1: Entwicklung von passiver und aktiver Sicherheit im Automobil[3]

[2] vgl. *Gleitende Leuchtweite* [Au10d, S. 9 ff.]

[3] grafische Darstellung aus [Le02, S. 92]

erwarten sind, bietet die aktive Fahrsicherheit noch immer sehr viel Entwicklungspotenzial, wie die **Abbildung 1.1** zeigt. Doch gerade die aktiven Sicherheitssysteme mit ihren ausgeklügelten und hochkomplexen Funktionen erfordern neben einer hohen Rechenleistung auch einen hohen Vernetzungsgrad zwischen den Steuergeräten. Während das ABS als eines der ersten aktiven Sicherheitssysteme im Fahrzeug noch auf ein einziges und autarkes Steuergerät beschränkt war, müssen z.B. mit der Einführung von ESP gleich mehrere Steuergeräte, u.a. für Bremse, Motor und Lenkung, kooperieren. Neue kamera- und radarbasierte Fahrerassistenten bis hin zum autonomen Fahren verursachen eine weitaus größere Kommunikationslast zwischen den Softwarekomponenten, für deren Bewältigung im Kapitel 5 - Interprozessorkommunikation, S. 73 ff. mögliche Lösungen vorgestellt werden.

Fehlerdiagnose. Ein weiterer Bereich in der Automobilelektronik beinhaltet die Diagnose von Fehlfunktionen. Die zunehmend komplexer werdenden Systeme machen es notwendig, teure Werkstattaufenthalte durch eine schnelle Fehlersuche zu verkürzen. In der Off-Board-Diagnose erkennt das Fahrzeug selbstständig einen auftretenden Defekt und empfiehlt der Werkstatt geeignete Maßnahmen zur Behebung des Fehlers. Darüberhinaus werden On-Board-Diagnosesysteme eingesetzt, um während der Fahrt z.B. die Einhaltung von Grenzwerten für den Schadstoffausstoß des Fahrzeugs zu überwachen[4].

Energiemanagement. Mit dem steigenden Anteil an Software und der dadurch erforderlichen höheren Rechenleistung im Fahrzeug rückt auch die Optimierung des Energieverbrauchs der datenverarbeitenden Systeme in den Fokus zukünftiger Entwicklungen in der E/E-Architektur. Da Rechner selbst keine nennenswerten Mengen an Energie speichern oder umwandeln, geht praktisch die gesamte aufgenommene elektrische Energie als Verlustleistung im Rechner „verloren". Vor diesem Hintergund zeigt das Kapitel 6 - Energieeffiziente Multiprozessorsysteme, S. 169 ff. verschiedene Möglichkeiten, mit denen die Verlustleistung in Rechensystemen signifikant reduziert werden kann.

Wertschöpfung und Kostendruck. Zukünftig wird es vermehrt darum gehen, die von Kunden und vom Gesetzgeber geforderte Funktionalität möglichst effizient zu realisieren. Durch den stark gestiegenen Anteil an der gesamten Wertschöpfung im Fahrzeug (vgl. **Abbildung 1.2**) ist die Automobilelektronik zusätzlich einem zunehmenden Kostendruck ausgesetzt. Diese Kosten beinhalten auf der Hersteller-

[4] mehr Informationen zu Diagnosesystemen im Fahrzeug in [Re09, S. 389 ff.]

seite die Entwicklungs- und Herstellungskosten, und auf der Kundenseite zählen Unterhalt- und Reparaturkosten. Für beide Seiten müssen deshalb technische Lösungen gefunden werden, welche die geforderte Funktionalität zu einem akzeptablen Preis realisieren können. In diesem Zusammenhang müssen eine ganze Reihe von Parametern und Rahmenbedingungen, wie die Systemintegration und die Portierbarkeit von modular organisierter Software, berücksichtigt werden, welche im Kapitel 2 - Rezentralisierung von Steuergeräten, S. 9 ff. dieser Arbeit näher beleuchtet werden.

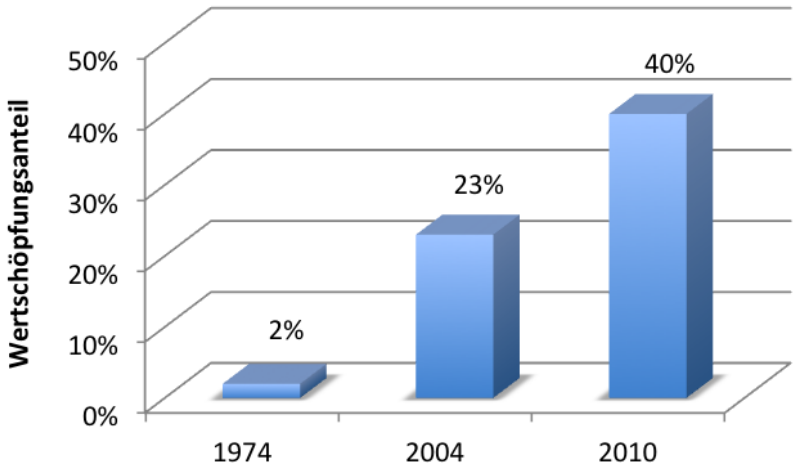


Abbildung 1.2: Steigerung des Anteils der Elektronik am Wert des Fahrzeug[5]

1.2 Zielsetzung

Problematik steigender Komplexität. Die E/E-Architektur moderner Fahrzeuge ist eine historisch gewachsene Topologie von verteilten und vernetzten Steuergeräten. Durch die Einführung zahlreicher neuer Funktionen befinden sich mittlerweile bis zu 100 Steuergeräte in einem einzigen Fahrzeug. Mit der Komplexität von neuen verteilten Anwendungen, wie im Fall der fahrerunterstützenden Systeme, steigt neben der Zahl der Steuergeräte auch der Vernetzungsgrad im Fahrzeug, wodurch die Kommunikationslast zwischen den Steuergeräten ebenfalls zunimmt. Das Herausforderung der vorliegenden Arbeit besteht darin, mit der gestiegenen Komplexität der Anwendungen umzugehen und die rechtzeitige Verarbeitung von Echtzeitdaten im Fahrzeug auch in der Zukunft durch ein deterministisches Verhalten des Rechensystems sicherzustellen.

[5] grafische Darstellung aus [Me06, S. 2]

Rezentralisierung. Für die zukünftige Entwicklung in der E/E-Architektur von Fahrzeugen gibt es grundsätzlich zwei verschiedene Ansätze: auf der einen Seite steht die bisherige Lösung durch ein dezentrales System aus verteilten Steuergeräten, und auf der anderen Seite steht die Rezentralisierung von Steuergeräten in einem oder einigen wenigen zentralen Rechenknoten. Die vorliegende Arbeit dient dazu, dem Ansatz der Rezentralisierung zu folgen und eine neuartige Rechnerarchitektur in Form des prototypisch implementierten Echtzeitparallelrechners *ConPar* zu entwerfen, um eine Rezentralisierung von Steuergeräten technisch umsetzen zu können.

1.3 Aufbau der Arbeit

Struktur und Inhalt. Die vorliegende Arbeit ist nachfolgend in sechs Kapitel aufgeteilt, welche sich mit der Beantwortung unterschiedlicher Fragestellungen beschäftigen, wobei die Kapitel inhaltlich aufeinander aufbauen.

Kapitel 2 - Rezentralisierung von Steuergeräten

- Wie ist der *derzeitige Stand* in der E/E-Architektur?
- Welche *Probleme* treten in der aktuellen E/E-Architektur auf?
- Könnten die Probleme durch *Rezentralisierung* gelöst werden?
- Wie würde eine rezentralisierte E/E-Architektur aussehen?

Kapitel 3 - Echtzeit- und Parallelrechner

- Was bedeutet Datenverarbeitung in Echtzeit?
- Welche besonderen Eigenschaften besitzen *Echtzeitrechner*?
- Wie sind *Parallelrechner* aufgebaut?
- Welche besonderen Eigenschaften besitzen *Parallelrechner*?
- Wie kann man einen *Echtzeitparallelrechner* konstruieren?

Kapitel 4 - Space-Sharing

- Auf welchem *Prinzip* basiert der Echtzeitparallelrechner?
- Wie sieht die *Architektur* des Echtzeitparallelrechners aus?
- Wie kann der Echtzeitparallelrechner *implementiert* werden?

Kapitel 5 - Interprozessorkommunikation

- Wie kann eine echtzeitfähige Kommunikation realisiert werden?
- Wie muss ein *Echtzeitverbindungsnetzwerk* aufgebaut sein?
- Wo liegen die Unterschiede zu bestehenden Verbindungsnetzwerken?
- Wie kann ein solches *Netzwerk on-Chip* implementiert werden?

Kapitel 6 - Energieeffiziente Multiprozessorsysteme

- Welche *Verlustleistungen* treten in einem Rechensystem auf?
- Welche Parameter *beeinflussen* die Verlustleistung?
- Wo liegen die *Einsparpotenziale* und wie hoch sind diese?
- Wie lässt sich der Energiebedarf im Rechensystem *minimieren*, ohne die Echtzeitfähigkeit zu verlieren?

Kapitel 7 - Integration in AUTOSAR

- Welche *Vorteile* bietet AUTOSAR?
- Mit welchem Aufwand kann AUTOSAR-konforme Software in einem Echtzeitparallelrechner *rezentralisiert* werden?
- Wie erfolgt die bisher busbasierte *Kommunikation* zwischen Steuergeräten im Echtzeitparallelrechner?
- Wie wird die *AUTOSAR-Methodik* eingebunden?

1.4 Veröffentlichungen

Publikationen. Einige Ergebnisse und Aspekte aus der vorliegenden Arbeit wurden von mir selbst an verschiedenen Stellen veröffentlicht. Die nachfolgende Auflistung zeigt die bereits erschienenen Veröffentlichungen nach Kapiteln geordnet.

Kapitel 2 - Rezentralisierung von Steuergeräten

[Au09b] Aust, S.; Richter, H.: *Parallel Computer Technology - A Solution for Automobiles? How car engineers can learn from parallel computing.* 3rd International Conference on Advanced Engineering Computing and Applications in Sciences; ADVCOMP 2009, Sliema, Oktober 2009, S. 148-152.

Kapitel 4 - Space Sharing

[Au10a] Aust, S.; Richter, H.: *Space Division of Processing Power for Feed Forward and Feed Back Control in Complex Production and Packaging Machinery*. World Automation Congress; WAC 2010, Kobe, September 2010, S. 1-6.

[Au10b] Aust, S.; Richter, H.: *Ein Echtzeitparallelrechner zur Rezentralisierung von Steuergeräten im Automobil*. erschienen in: Tschöke, H.; Krahl, J.; Munack, A. (Hrsg.): *Innovative Automobiltechnik II*. Expert Verlag, 2010, S. 70-88.

Kapitel 5 - Interprozessorkommunikation

[Au10c] Aust, S.; Richter, H.: *Real-time Processor Interconnection Network for FPGA-based Multiprocessor System-on-Chip (MPSoC)*. 4th International Conference on Advanced Engineering Computing and Applications in Sciences; ADVCOMP 2010, Florenz, Oktober 2010, S. 47-52.

[Au11a] Aust, S.; Richter, H.: *Skalierbare Rechensysteme für Echtzeitanwendungen*. erschienen in: Halang, W. A. (Hrsg.): *Herausforderungen durch Echtzeitbetrieb.* Springer Verlag, 2011, S. 111-120.

Kapitel 6 - Energieeffiziente Multiprozessorsysteme

[Au11b] Aust, S.; Richter, H.: *Energy-Aware MPSoC with Space-Sharing for Real-Time Applications*. 5th International Conference on Advanced Engineering Computing and Applications in Sciences; ADVCOMP 2011, Lissabon, November 2011, S. 54-59.

[Au12] Aust, S.; Richter, H.: *Energy-aware MPSoC for Real-time Applications with Space-Sharing, Adaptive and Selective Clocking and Software-first Design*. International Journal on Advances in Software, Vol. 5, Nr. 3 & 4, 2012, S. 368-377.

Kapitel 2 - Rezentralisierung von Steuergeräten

Inhalt. Die Anzahl und die Komplexität von elektronischen Steuergeräten im Automobil steigen seit Jahren dramatisch an, wodurch eine ganze Reihe vielfältiger neuer Probleme entsteht[1]. Eine mögliche Lösung für die gegenwärtig und zukünftig anstehenden Probleme könnte in der Rezentralisierung von Steuergeräten in einem Echtzeitparallelrechner liegen. Das Kapitel analysiert einige drängende Probleme der Automobilelektronik und zeigt auf, ob und wie diese Probleme durch Rezentralisierung von Steuergeräten gelöst werden könnten.

2.1 Bordnetzarchitektur

2.1.1 Dezentrale Topologie

Historisch gewachsene Topologie. Die derzeitige Architektur der Bordnetze in Fahrzeugen entspricht einer dezentralen Topologie aus Steuergeräten, welche über Kommunikationsbusse miteinander vernetzt sind. Die heutige Topologie ist nicht geplant, sie ist in der Historie sukzessiv gewachsen. So beschränkte sich die Elektronik im Auto in den 1980er Jahren noch auf einige wenige Steuergeräte und die Anzahl der Signalleitungen konnte mit der Einführung von CAN stark reduziert werden[2]. In den folgenden Jahren wuchs die Zahl der Steuergeräte jedoch kontinuierlich an. So besitzt die aktuelle Generation des Audi A8 bis zu 95 Steuergeräte[3]. Gleichzeitig erhöhte sich über die Jahre die funktionale Komplexität und damit der Vernetzungsgrad zwischen den Steuergeräten, so dass für die steuergeräteübergreifende Kommunikation heute mehrere verschiedene Bussysteme in einem einzigen Fahrzeug koexistieren. Eine ausführliche Beschreibung der verschiedenen Bussysteme im Fahrzeug befindet sich in [No06, S. 64 ff.].

Funktionsorientierte Kommunikation. Die **Abbildung 2.1** zeigt das beispielhafte Schema einer Bordnetzarchitektur. Die Kommunikation im Fahrzeug ist nach Funktionsdomänen (z.B. Antrieb, Komfort und Infotainment) organisiert. Topologische Gegebenheiten wie die räumliche Anordnung der Steuergeräte im Fahrzeug bleiben aufgrund der i.d.R. auf Funktionsdomänen abgegrenzten Kommunikations-

[1] vgl. [Au09b]

[2] vgl. [No06, S. 47]

[3] Statistik der Audi AG, Stand: 2010 [Au09a, S. 2]

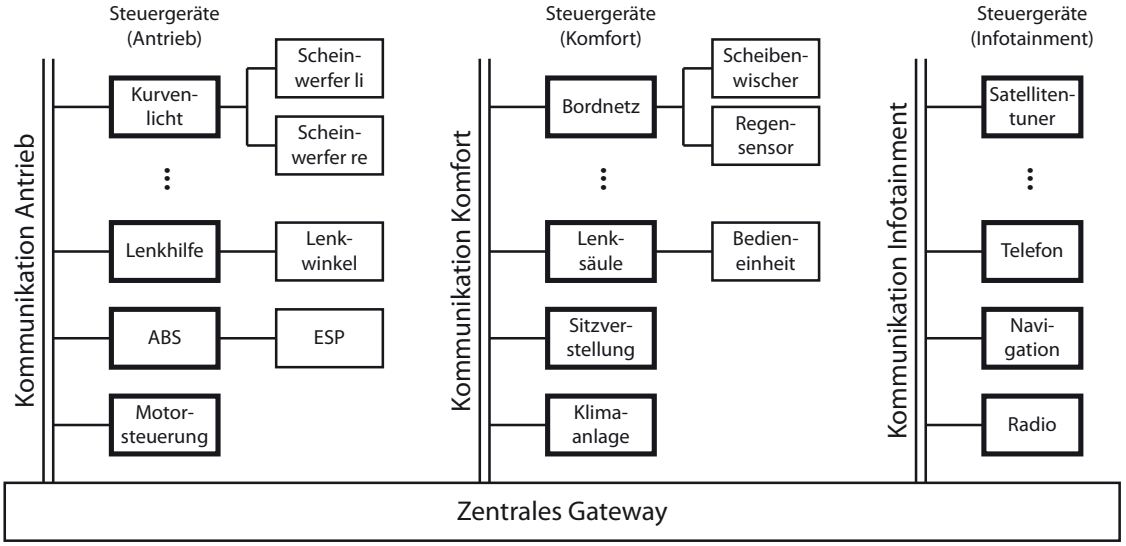


Abbildung 2.1: Dezentrale Bordnetzarchitektur[4]

last weitgehend unberücksichtigt. Die Kommunikation zu externen Systemen ist durch ein zentrales Gateway realisiert, welches auch zur Offboard-Diagnose des Fahrzeugs bei Werkstattaufenthalten dient. Die Kombination aus funktionsorientierter Kommunikation und lokaler hardwarenaher Platzierung der Steuergeräte im Fahrzeug führt zu einem weit verzweigtem Geflecht aus Kommunikationsverbindungen, dem heutigen Kabelbaum im Auto.

2.1.2 Zentralisierte Topologie

Zentrales Steuergerät mit peripheren Sensor-/Aktorknoten. Durch eine Rezentralisierung der Steuergeräte wird die gesamte Funktionssoftware des Fahrzeugs in nur wenigen Steuergeräten bzw. Rechenknoten zusammengefasst. Die Kommunikation der zentralen Rechenknoten mit der Peripherie im Fahrzeug erfolgt über Sensor-/Aktorknoten. Dabei werden Sensoren und Aktoren nach ihrer Lokalität im Fahrzeug zu Kommunikationsgruppen zusammengefasst. Durch die Berücksichtigung der topologischen Gegegenheiten im Fahrzeug ließe sich die Länge der Leitungen im Kabelbaum auf ein Minimum reduzieren.

Ortsbezogene periphere Kommunikation. Die **Abbildung 2.2** zeigt beispielhaft den Aufbau einer zentralisierten Bordnetzarchitektur. Den Mittelpunkt dieser Architektur bildet ein zentrales Steuergerät, das für die Steuerung des gesamten Fahrzeugs verantwortlich ist. Die Kommunikation zwischen dem zentralen Steuergerät und

[4] Darstellung abgeleitet aus dem Vernetzungskonzept des VW Golf VI [Vw08, S. 66 f.]

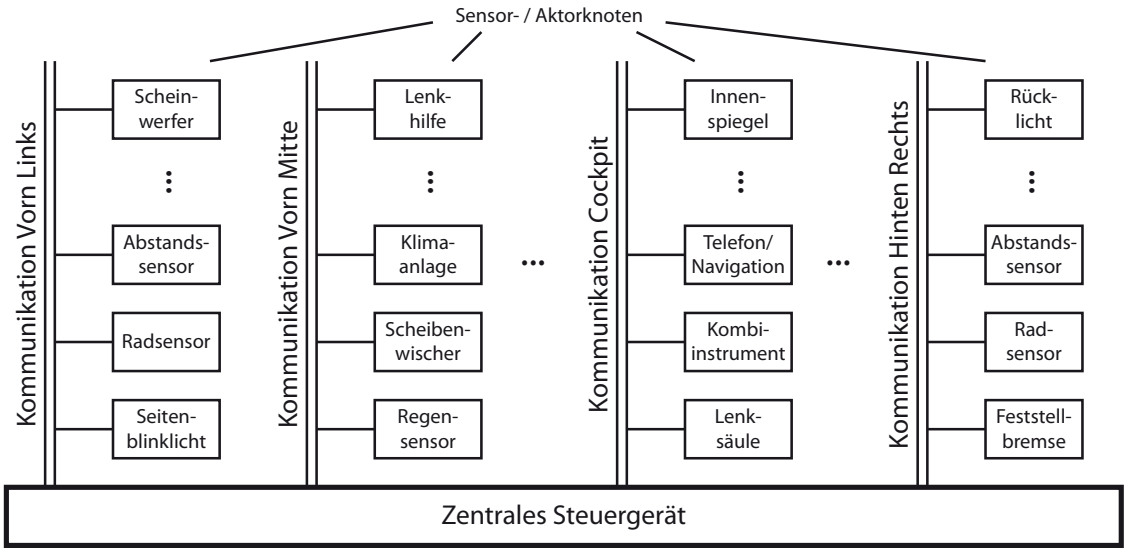


Abbildung 2.2: Rezentralisierte Bordnetzarchitektur

der Peripherie im Fahrzeug erfolgt ortsbezogen, d.h. die Bordnetzarchitektur wird an den lokalen Gegebenheiten im Fahrzeug ausgerichtet. Eine solche Bordnetzarchitektur erfordert den Einsatz einer homogenen Kommunikationslösung, wie das in [Wi09, S. 72 ff.] beschriebene Echtzeit-Kommunikationsnetzwerk *CarRing II*.

2.2 Pro und Kontra Rezentralisierung

Inhalt. Für eine Vielzahl der gegenwärtig und zukünftig anstehenden Probleme in der Automobilelektronik könnte die Rezentralisierung von Steuergeräten eine mögliche Lösung bieten. In diesem Abschnitt werden einige dieser Probleme näher beleuchtet und deren Lösung durch eine Rezentralisierung aufgezeigt.

2.2.1 Kommunikation

Hoher Vernetzungsgrad. Kommunikation spielt in der Automobilelektronik eine immer größer werdende Rolle. Der Hauptgrund dafür liegt u.a. darin, dass viele neue Funktionen, die z.B. zur aktiven Fahrsicherheit beitragen, nur noch durch den kooperativen Betrieb mehrerer Steuergeräte realisierbar sind. Das seit dem *Elchtest* im Jahr 1997 serienmäßig eingeführte System zur Kontrolle der Fahrdynamik (ESC) wirkt beispielweise auf Motor und Bremse gleichermaßen, um das Fahrzeug mittels Antiblockiersystem (ABS) und Antischlupfregelung (ASR) in der Längsdynamik sowie mittels Giermomentenregelung (GMR) in der Querdynamik zu stabilisieren. Bei komplexen verteilten Funktionen, wie dem als Abstandstem-

pomat bekannten Adaptive Cruise Control (ACC), sind mehr als 20 Steuergeräte involviert [Ra10, S. 9]. Neue komplexe Funktionen und die gestiegene Anzahl an Steuergeräten (derzeit ≤ 100) führen zu einem extrem hohen Vernetzungsgrad im Fahrzeug. Die **Tabelle 2.1** und die **Tabelle 2.2** zeigen die historische Entwicklung in der E/E-Architektur am Zahlenbeispiel des VW Golf und der Mercedes E-Klasse.

Tabelle 2.1: Entwicklung der Automobilelektronik im VW Golf[5]

Plattform	Golf IV (1998)	Golf V (2003)	Golf VI (2010)
Steuergeräte (ECU)	17	35	49
CAN-Nachrichten	147	307	704
CAN-Signale	434	2669	6516

Tabelle 2.2: Entwicklung der Automobilelektronik in der Mercedes E-Klasse[6]

Plattform	W123 (-1985)	W124 (-1995)	W210 (-2002)	W211 (-2008)	W212 (-2009)
Steuergeräte (ECU)	2	7	30	52	67
CAN-Signale	-	-	≈ 200	≈ 4100	≈ 6000

Heterogene Kommunikation. Neben der steigenden Kommunikationslast liegt ein weiteres Problem in der Heterogenität der Kommunikation aufgrund unterschiedlicher Aufgaben. Während der im Jahre 1980 vorgestellte *CAN-Bus* über einen relativ langen Zeitraum von etwa 20 Jahren völlig ausreichend war, haben sich in den letzten Jahren zusätzlich auch andere Bussysteme wie *FlexRay* und *MOST* für die Übertragung von Daten mit harten Echtzeitanforderungen bzw. für die Übertragung von Multimedia- und Infotainmentdaten fest im Fahrzeug etabliert[7]. Dies hat dazu geführt, dass heute in einem modernen Fahrzeug mehrere und zueinander inkompatible Kommunikationssysteme koexistieren. Einige Beispiele für die verschiedenen Systeme sind in der **Tabelle 2.3** aufgeführt. Aufgrund der stark ausgeprägten Heterogenität und den damit verbundenen Kompatibiltätsproblemen in der Kommunikation wird seit einigen Jahren an verschiedenen Stellen über die Einführung von IP-basierten Bordnetzen im Fahrzeug diskutiert[8,9].

[5] statistische Daten aus *The Volkswagen PQ architecture* [Ra10, S. 8]

[6] statistische Daten aus *Impact on Networking and Communication* [Bo10, S. 6]

[7] vgl. FlexRay [Zi08, S. 64 ff.] und MOST [Zi08, S. 84 ff.]

[8] eine ausführliche Betrachtung von Ethernet in Fahrzeugen befindet sich in [Br10]

[9] Beispielprojekt: SEIS - Sicherheit in Eingebetteten IP-basierten Systemen, Förderprojekt des Bundesministeriums für Bildung und Forschung, Laufzeit Juli 2009 - Juni 2012

Tabelle 2.3: Übersicht verschiedener Kommunikationsmedien im Fahrzeug[10]

Bussystem	Max. Datenrate	Anwendungsbereich
K-Line	10,4 kbit/s	Diagnoseaufgaben
SAE J1850	41,6 kbit/s	On-/ Offboard Kommunikation
CAN	1 Mbit/s	Low / High Speed Anwendungen
LIN	19,2 kbit/s	Sensor-Aktor Anwendungen
FlexRay	10 Mbit/s	X-by-wire
MOST	150 Mbit/s	Telematik und Multimedia

Bordnetz-Architektur. Die große Zahl an Steuergeräten und ein hoher Vernetzungsgrad zwischen den Steuergeräten führen zwangsläufig zu der sehr komplexen Bordnetz-Architektur moderner Fahrzeuge. Aufgrund der gestiegenen Kommunikationslast koexistieren heute mehrere CAN-Busse, getrennt nach Einsatzbereichen wie Antrieb oder Komfort. Zusätzlich dazu werden Aufgaben, für die der CAN-Bus ungeeignet ist, durch weitere Bussysteme wie FlexRay oder MOST abgedeckt. Insgesamt entsteht so ein weit verzweigtes Bordnetz, welches eine ganze Reihe von Problemen, z.B. bei der Konstruktion und der Fertigung, mit sich bringt. So kann der Kabelbaum im Auto bis heute nicht vollständig automatisiert gefertigt werden, wodurch allein dessen Fertigungszeit einen permanent steigenden Kostenfaktor darstellt, welcher als unmittelbare Konsequenz aus der steigenden Anzahl der Steuergeräte und deren Vernetzung im Fahrzeug resultiert. Darüber hinaus werden zur Herstellung des Kabelbaums große Mengen an wertvollem Kupfer benötigt, was sich im Materialpreis, aber auch im Gesamtgewicht des Bordnetzes widerspiegelt. Mit bis zu *1500 Einzelleitungen* und *50 kg Gewicht* beim Audi A8[11] gilt das Bordnetz heute als „eines der größten, schwersten und teuersten Bauteile im Automobil“ [Au09a, S. 2]. Zusätzlich vergrößert sich durch die wachsende Zahl unterschiedlichster Steckverbindungen auch das Risiko von technischen Defekten in der Bordelektronik, welche z.B. aufgrund von Kontaktproblemen, hervorgerufen durch Materialermüdung oder Oxidation, beim späteren Betrieb des Fahrzeugs auftreten können. Hier können vor allem sporadische Fehler auftreten, die sich nur sehr schwer lokalisieren lassen. In der **Tabelle 2.4** sind einige wichtige Kenngrößen des Kabelbaums im Auto mit Zahlen belegt, die einen Durchschnittswert über alle Fahrzeugklassen darstellen.

[10] Daten aus [Zi08]

[11] Statistik der AUDI AG, Stand: 2010 [Au09a, S. 2]

Tabelle 2.4: Kenndaten eines durchschnittlichen Kabelbaums[12]

Elektrische Verbindungen	> 700
Gesamtleitungslänge	> 2 km
Anzahl Stecker	> 200
Fertigungsminuten	ca. 900 min
Gewicht	25 kg

Verlagerung auf interne Kommunikation. Die wachsende Zahl an Steuergeräten und der hohe funktionale Vernetzungsgrad führen zu einem steigenden Kommunikationsbedarf zwischen den Steuergeräten, welcher derzeit durch den Einsatz mehrerer unterschiedlicher Bussysteme abgedeckt wird, was wiederum zu einer komplexen Bordnetz-Architektur führt. Die Rezentralisierung der Steuergeräte in einem Echtzeitparallelrechner könnte die steuergeräteübergreifende Kommunikation auf eine geräteinterne Kommunikation verlagern, was den Kabelbaum im Fahrzeug deutlich vereinfacht.

2.2.2 Systemintegration

Restbussimulation. Eine große Herausforderung bei der Entwicklung von Steuergeräten ist die Integration der Steuergeräte in das umgebende System im Fahrzeug. Da zum Zeitpunkt der Entwicklung i.d.R. noch kein reales Fahrzeug zur Verfügung steht, muss das Steuergerät zunächst mit Hilfe einer umfangreichen Restbussimulation gemäß einer Spezifikation getestet werden, in der das gesamte Fahrzeug und dessen Steuergeräte abgebildet werden. Dies stellt, auch aufgrund der Vielzahl der Steuergeräte und des hohen Vernetzungsgrades, einen erheblichen zusätzlichen Zeit- und Kostenfaktor bei der Entwicklungsarbeit dar.

Integrationstest. Die funktionale Komplexität moderner Automobilelektronik stellt auch für den Integrationstest der Steuergeräte im realen Fahrzeug einen hohen Kostenfaktor dar. Auf der einen Seite muss die korrekte Funktion des Systems durch umfangreiche Testszenarien nachgewiesen sein und auf der anderen Seite wird bei einem Fehlverhalten die Suche nach der Ursache durch die Komplexität des Systems erschwert.

[12] Statistik der BMW AG, Stand: 2009 [We09]

Verbesserte Systemintegration durch Rezentralisierung. Die Rezentralisierung der Steuergerätesoftware in einem Echtzeitparallelrechner beschränkt den Vorgang der Entwicklung und der Systemintegration auf die Ebene der Funktionssoftware. Vorraussetzung hierfür ist die Portierbarkeit von Funktionssoftware, welche mit der Einführung von AUTOSAR jedoch mit eher geringen Mitteln realisierbar ist[13]. Mit der Verlagerung der steuergeräteübergreifenden Kommunikation auf eine rechnerinterne Interprozessorkommunikation durch ein echtzeitfähiges Verbindungsnetzwerk[14] entfällt die Notwendigkeit von Simulation und Test der bisherigen externen Kommunikationswege auf der Basis heterogener Bussysteme[15] wie CAN und FlexRay zwischen den Steuergeräten.

2.2.3 Qualitätssicherung

Qualität der Hardware. Die Vielzahl an Steuergeräten und die komplexe Architektur des Kabelbaums (inkl. Steckverbindungen etc.) im Fahrzeug erhöhen signifikant die Wahrscheinlichkeit, dass eines der verwendeten Bauteile aufgrund eines technischen Defektes ausfällt, was u.U. zur Verärgerung von Kunden durch die Nichtverfügbarkeit des Fahrzeugs und zu kostenintensiven Werkstattaufenthalten führt. Aus diesem Grund muss das Ziel sein, die Anzahl der Steuergeräte und deren Verkabelungsaufwand auf ein Minimum zu reduzieren.

Qualität der Software. Aufgrund des hohen funktionalen Vernetzungsgrades im Fahrzeug ist die Kooperation der Software verschiedener Steuergeräte von großer Bedeutung für die spätere Funktion des Fahrzeugs. Mit der Komplexität wächst jedoch die Gefahr, dass bei der Systemintegration im realen Fahrzeug aufgrund der steuergeräteübergreifenden Kommunikation nicht spezifizierte Situationen auftreten, die ein fehlerhaftes Verhalten des Fahrzeugs verursachen könnten.

Bessere Testmöglichkeiten auf einem einzigen Rechner. In einem Parallelrechner beschränkt sich die Systemintegration der Funktionssoftware auf Module, deren korrektes Verhalten auf einem einzigen Rechner besser getestet und überwacht werden kann als in einem dezentralen Verbund verschiedenster Steuergeräte. Dadurch würde die Verifikation und Fehleranalyse des Systems vereinfacht.

---

[13] vgl. Kapitel 7 - Integration in AUTOSAR, S. 198 ff.

[14] vgl. Kapitel 5 - Interprozessorkommunikation, S. 73 ff.

[15] vgl. Tabelle 2.3: Übersicht verschiedener Kommunikationsmedien im Fahrzeug, S. 13

2.2.4 Modularität und Skalierbarkeit

Breite Modell- und Ausstattungspalette. Das Angebot in der Modell- und Ausstattungspalette von Fahrzeugen wurde in den letzten beiden Dekaden herstellerübergreifend extrem erweitert. Diesen Sachverhalt verdeutlicht die **Tabelle 2.5** mit der Gegenüberstellung der Modellpaletten einiger führender Automobilhersteller aus den Jahren 1980 und 2010. Die Gründe für die gewachsene Angebotsvielfalt finden sich hauptsächlich in Vertrieb und Marketing der Automobilhersteller, wo möglichst viele individuelle Kundenwünsche abgedeckt werden sollen, um einen großen Kundenkreis zu erreichen. Um die Entwicklungskosten trotzdem gering zu halten, müssen einmal entwickelte und getestete Komponenten auf unterschiedlichsten Plattformen, vom Kleinwagen bis zur Luxuslimousine, eingesetzt werden können. Dieses Ziel kann nur durch Standardisierung und die konsequente Umsetzung von Eigenschaften wie Modularität, Portierbarkeit und Skalierbarkeit erreicht werden.

Modularität und Skalierbarkeit im Parallelrechner. In der heutigen dezentralen E/E-Architektur der Fahrzeuge ist die Funktionssoftware und damit die erlebbare Funktionalität auf verschiedene Steuergeräte verteilt[16]. Dagegen wird bei einer Rezentralisierung im Echtzeitparallelrechner eine homogene Rechnerarchitektur verwendet, deren Rechenleistung entsprechend den Anforderungen skaliert werden kann. Auf diese Weise kann die Funktionalität des Fahrzeugs je nach Ausstattungsvariante durch das Hinzufügen oder Entfernen von modularer Funktionssoftware verändert werden[17], wodurch die geforderte Modellvielfalt mit einem Echtzeitparallelrechner sehr viel besser als bisher realisiert werden kann.

Tabelle 2.5: PKW-Modellpalette führender Automobilhersteller[18]

Hersteller	**Angebotene Modelle**	
	im Jahr 1980	**im Jahr 2010**
Audi	80, 100, 200	A1, A3, A4, A5, A6, A7, A8, Q3, Q5, Q7, R8, TT
BMW	3er, 5er, 6er, 7er, M1	1er, 3er, 5er, 6er, 7er, X1, X3, X5, X6, Z4

[16] vgl. 2.1.1 Dezentrale Topologie, S. 9 f.

[17] vgl. Kapitel 4 - Space-Sharing, S. 46 ff.

[18] ohne Berücksichtigung verschiedener Karosserieformen, Motorisierungen und Ausstattungen

Hersteller	Angebotene Modelle	
	im Jahr 1980	im Jahr 2010
Daimler	E, S, SL, G	A, B, C, CLK, E, R, CLS, S, CL, SLK, SL, GLK, M, GL
Volkswagen	Polo, Derby, Golf, Jetta, Passat, Scirocco, Caddy	Fox, Polo, Beetle, Golf, Jetta, Passat, Passat CC, Phaeton, Scirocco, Eos, Golf Plus, Touran, Sharan, Caddy, Tiguan, Amarok, Touareg

2.2.5 Portierbarkeit

Kürzere Modelllaufzeiten. Ein weiterer wichtiger Punkt bei der Entwicklung von Steuergeräten sind die immer kürzer werdenden Produktionszyklen einer Modellgeneration, wie die **Abbildung 2.3** zeigt. Dieser Trend bedeutet für die Entwicklungsarbeit, dass neue Modelle nicht mehr von Grund auf neu konstruiert werden können. Vielmehr müssen einzelne Komponenten, die für den Einsatz in neuen Modellen keiner Veränderung bedürfen, ohne Anpassungen vom Vorgänger übernommen werden. Aus diesem Grund muss bei der Entwicklung von Automobilelektronik ganz wesentlich auch auf die Wiederverwendbarkeit und Portierbarkeit von Hard- und Softwarekomponenten geachtet werden.

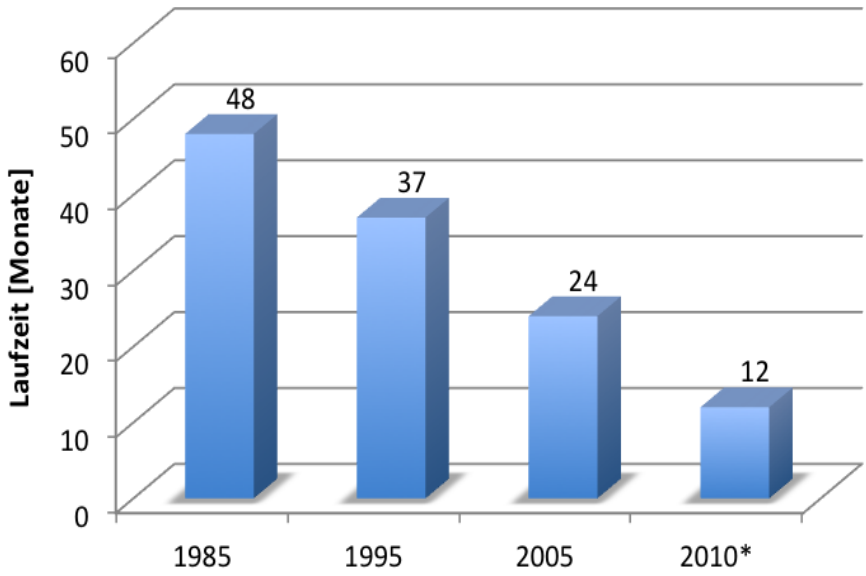


Abbildung 2.3: Verringerung der Laufzeiten von Fahrzeugmodellen[19] [20]

[19] grafische Darstellung aus [Me06, S. 2]

[20] * geschätzter Wert

Einführung von AUTOSAR. Ein wichtiger Schritt in Richtung Portierbarkeit ist die herstellerübergreifende Einführung von AUTOSAR[21]. Durch die Standardisierung eines geschichteten Softwaremodells erfolgt die strikte Trennung der Funktionssoftware von der ausführenden Hardware, was die Portierbarkeit bestehender Funktionssoftware auf die Hardware nachfolgender Generationen erlaubt[22]. Durch die Separation der Funktionssoftware und die Standardisierung von Schnittstellen ermöglicht AUTOSAR aber auch die Rezentralisierung der Funktionalität von existierender Steuergerätesoftware in einem Echtzeitparallelrechner, ohne die Steuergeräte selbst aufwändig emulieren zu müssen.

2.2.6 Langzeitverfügbarkeit

Nutzungsdauer von Automobilen. Entscheidend für die rechtliche Verpflichtung zur Lieferung von Ersatzteilen ist der übliche Nutzungszeitraum, der bei Automobilen mit ca. 10-15 Jahren angegeben wird [Ha03, S. 14]. Das Alter der Fahrzeuge bei der Stilllegung beträgt in Deutschland im Schnitt etwa 12 Jahre[23], wie die **Abbildung 2.4** zeigt. Das durchschnittliche Fahrzeugalter ist im Zeitraum der Jahre 2000 bis 2011 von 6,9 auf 8,3 Jahre gestiegen, weshalb sich der übliche Nutzungszeitraum

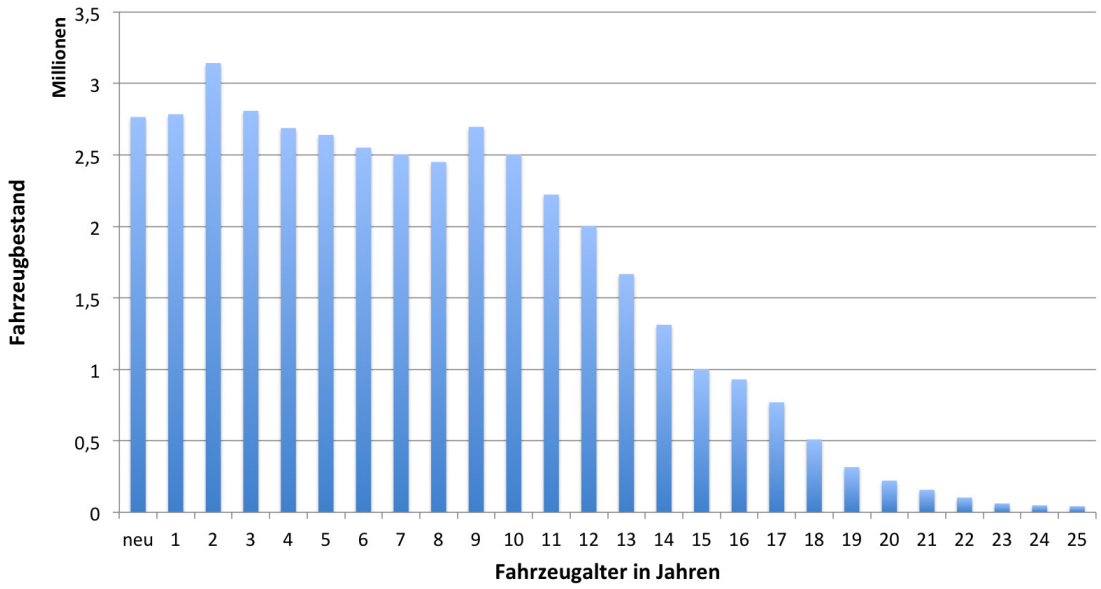


Abbildung 2.4: Fahrzeugbestand nach Fahrzeugalter in Deutschland[24]

[21] AUTomotive Open System ARchitecture, www.autosar.org

[22] vgl. Kapitel 7 - Integration in AUTOSAR, S. 198 ff.

[23] Statistik des Kraftfahrt-Bundesamtes, Stand: April 2011 [Th11, S. 8]

[24] Statistik des Kraftfahrt-Bundesamtes, Stand: 2009 [Kb09, S. 7]

von Fahrzeugen in der Zukunft wahrscheinlich in die Richtung von 15 Jahren bewegen wird. Das bedeutet, dass für diesen Zeitraum Ersatzteile, auch für die Elektronik des Fahrzeugs, lieferbar sein müssen, wobei die Frist erst mit dem Auslaufen einer Modellreihe beginnt.

Produktionszyklen von Halbleitern. Gegenüber der Nutzungsdauer von Automobilen stellt sich die Produktion von Halbleitern als ein eher kurzfristiges Ereignis dar. Nach der als Moore`sches Gesetz[25] bekannten Regel verdoppelt sich die Integrationsdichte von integrierten Schaltungen etwa alle 18 bis 24 Monate. Da der Anteil der Automobilelektronik am weltweiten Halbleitermarkt mit etwa 8 Prozent relativ gering ausfällt, ist es für einen Halbleiterhersteller unmöglich, eine moderne Fab ausschließlich für die Produktion von Automobilelektronik rentabel zu betreiben. Eine Nachserienfertigung wird dadurch praktisch ausgeschlossen. Die daraus zwangsläufig entstehende Verknappung bzw. Nichtverfügbarkeit älterer elektronischer Steuergeräte ist heute bereits Realität [Do05]. Der einzige derzeit verfügbare Ausweg besteht in der Langzeitlagerung von elektronischen Baugruppen gemäß einer sehr langfristigen Bedarfseinschätzung, wobei für die Lagerung von elektronischen Baugruppen über einen Zeitraum von mehr als 15 Jahren nicht genügend Erfahrungswerte existieren [Zv02, S. 19].

Herausforderungen für Hersteller und Lieferanten. Die zeitliche Diskrepanz zwischen der Nutzungsdauer von Automobilen und den Produktionszyklen der Halbleiterindustrie (vgl. **Abbildung 2.5**) macht eine Langzeitversorgung mit elektronischen Ersatzteilen schwierig. Der Anteil der Automobilelektronik am Wert des Fahrzeugs ist in den letzten zehn Jahren zudem dramatisch gestiegen, was sich auch in der Anzahl der Steuergeräte pro Fahrzeug widerspiegelt[26]. Gleichzeitig ist die Modellvielfalt herstellerübergreifend rapide gewachsen[27]. Diese Situation führt insgesamt zu einer extremen Herausforderung bei der Bevorratung mit elektronischen Ersatzteilen durch die Hersteller. Eine mögliche Lösung des Problems liegt in der Portierbarkeit von Hard- und Software[28]. Die Portierbarkeit von Funktionssoftware auf unterschiedliche Hardware bietet die Möglichkeit, elektronische Bauelemente oder -gruppen nachfolgender Modellgenerationen für die Ersatzteilversorgung bereits ausgelaufener Modellreihen zu nutzen.

[25] Gordon Moore (geb. 1929), Mitbegründer des Chipherstellers Intel [Mo65]

[26] vgl. Tabelle 2.1: Entwicklung der Automobilelektronik im VW Golf, S. 12

[27] vgl. Tabelle 2.5: PKW-Modellpalette führender Automobilhersteller, S. 16

[28] vgl. 2.2.5 Portierbarkeit, S. 17 f.

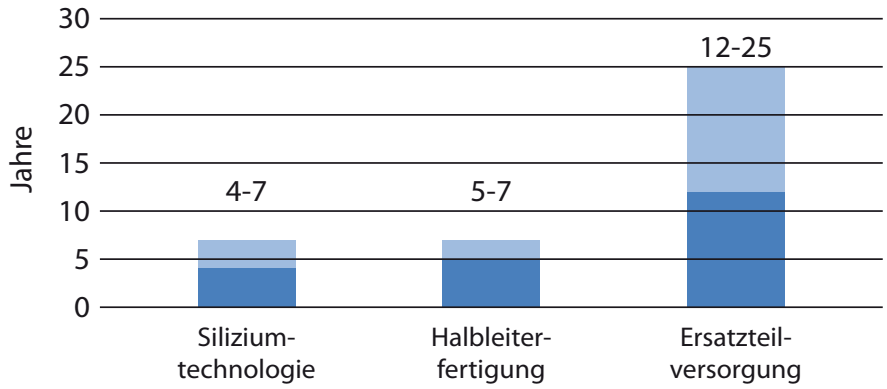


Abbildung 2.5: Laufzeiten bei Automobilelektronik[29]

2.2.7 Single Point of Failure

Problem des Single Point of Failure. Den größten Schwachpunkt bei der Rezentralisierung eines verteilten Systems aus Steuergeräten stellt das Problem des *Single Point of Failure* (SPOF) dar. Während der Ausfall eines Steuergerätes in einem verteilten System nur begrenzte Auswirkungen besitzt und i.d.R. vom umgebenden System abgefangen werden kann, so hätte bei einer vollständigen Rezentralisierung der Ausfall des zentralen Steuergerätes katastrophale Auswirkungen auf das gesamte Fahrzeug. Ein solcher Totalausfall ist insbesondere bei sicherheitsrelevanten Funktionen nicht zu akzeptieren, so dass eine vollständige Rezentralisierung der Funktionssoftware auf ein einzelnes zentrales Steuergerät als nicht praxistauglich zu bewerten ist. An dieser Stelle muss ein Kompromiss zwischen der Rezentralisierung auf der einen Seite und dem Risiko eines *Single Point of Failure* auf der anderen Seite gefunden werden[30].

2.3 Das Funktionsmodell von ConPar

2.3.1 Vollständige Rezentralisierung

Rezentralisierung von Funktionssoftware. Um das verteilte System an Steuergeräten im Fahrzeug zu rezentralisieren, muss die Funktionssoftware der Steuergeräte an zentraler Stelle zusammengeführt werden. Die **Abbildung 2.6** zeigt das Funktionsmodell bei einer vollständigen Rezentralisierung der Steuergeräte. Den Kern

[29] grafische Darstellung auf Basis von [Th11, S. 8] und [Zv02, S. 15]

[30] vgl. 2.3.2 Domänenorientierte Rezentralisierung, S. 22 f.

(grün dargestellt) bilden die einzelnen Funktionen wie z.B. Fahrerassistenten, die in das Fahrzeug integriert werden sollen und aus denen sich die Eigenschaften des Fahrzeugs definieren. Dieser Teil ist für den Nutzer von außen als spätere Funktion sichtbar und erlebbar. Erreicht wird die Funktionalität des Fahrzeugs durch die Steuerung der einzelnen Fahrzeugkomponenten, wie z.B. dem Motor, der Lenkung oder der Bremse. Diese Steuerelemente sind in dem gelb markierten Bereich eingetragen. Der gelbe Bereich bildet zusammen mit dem grün dargestellten Funktionskern den zentralen Rechner im Fahrzeug. Die Kommunikation des Zentralrechners mit den peripheren Sensoren und Aktoren des Fahrzeugs erfolgt über ein fahrzeugweites Kommunikationsnetzwerk (rot dargestellt). Die äußere blaue Hülle repräsentiert die Sensoren und Aktoren, welche Daten aus der realen Umgebung sammeln und die Steuerbefehle des Zentralrechners in Aktionen umsetzen.

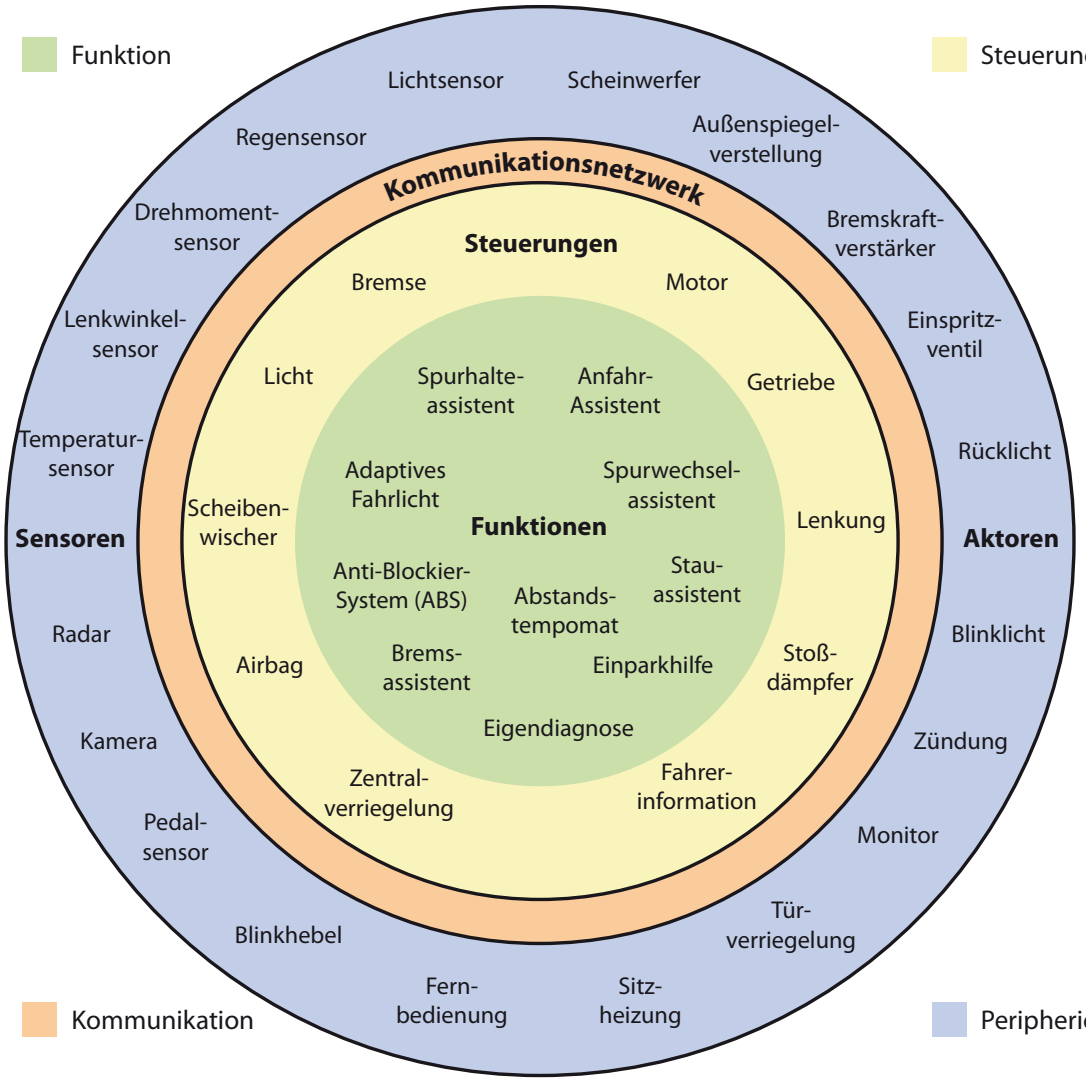


Abbildung 2.6: Vollständige Rezentralisierung von Steuergeräten

2.3.2 Domänenorientierte Rezentralisierung

Funktionsorientierte Rezentralisierung. In der E/E-Architektur eines modernen Fahrzeugs ist die Kommunikation zwischen den Steuergeräten i.d.R. durch mehrere Bussysteme realisiert. Die Datenbusse sind über ein zentrales Gateway miteinander gekoppelt, so dass auch eine busübergreifende Kommunikation der Steuergeräte möglich ist. Der Hintergrund dieser Architektur besteht darin, dass ein einziger Datenbus nicht mehr in der Lage ist, die gesamte Kommunikationslast im Fahrzeug abzudecken. Aus diesem Grund wird die Kommunikationslast auf mehrere verschiedene Bussysteme aufgeteilt. Da Steuergeräte häufig mit Steuergeräten derselben Funktionsdomäne kommunizieren, erfolgt die Aufteilung der

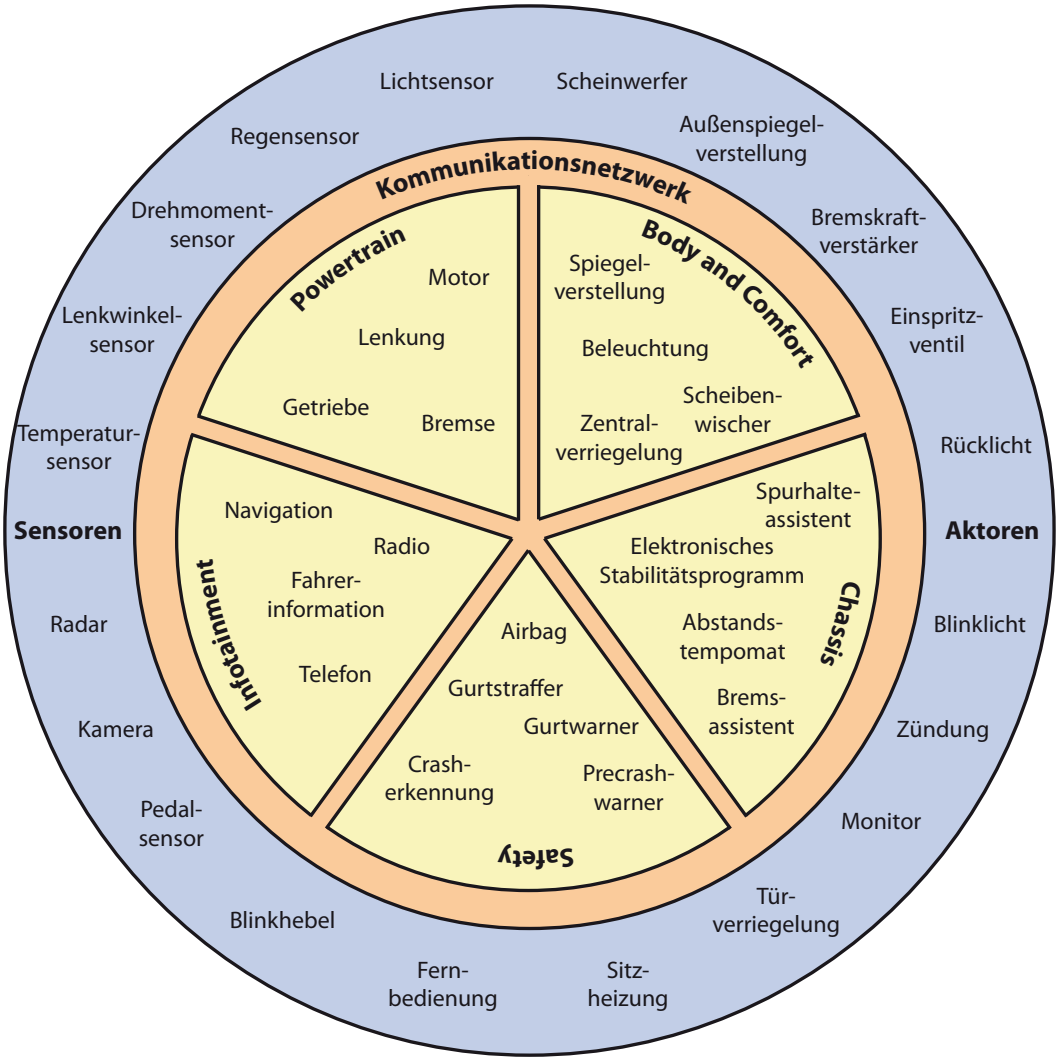


Abbildung 2.7: Funktionsorientierte Rezentralisierung der Steuergeräte[31]

[31] Aufteilung nach Domänen auf Basis der Application Interfaces in AUTOSAR Release 4.0

Kommunikationslast funktionsorientiert, so dass die Steuergeräte innerhalb einer Funktionsdomäne auch über einen gemeinsamen Bus kommunizieren. In der **Abbildung 2.7** ist beispielhaft die Rezentralisierung in die fünf Funktionsdomänen *Powertrain*, *Body and Comfort*, *Chassis*, *Safety* und *Infotainment* dargestellt. Innerhalb dieser Funktionsdomänen findet die meiste Kommunikation statt, so dass der größte Teil der Inter-ECU-Kommunikation auf eine interne Kommunikation im rezentralisierten Steuergerät abgebildet werden kann. Für das Ziel der Rezentralisierung von Steuergeräten stellt diese Aufteilung einen Kompromiss dar zwischen der Verlagerung der bisherigen Buskommunikation auf eine steuergeräteinterne Kommunikation und der Entschärfung des Problems des *Single Point of Failure*. Zusätzlich müssen sicherheitskritische Funktionen in redundante Systeme mit geeigneten Notlaufeigenschaften verlagert werden. Auf die Betrachtung des *Single Point of Failure* und daraus folgenden Konsequenzen bei sicherheitskritischen Funktionen soll jedoch aufgrund der Komplexität des Problems an dieser Stelle nicht weiter eingegangen werden.

2.3.3 Konfigurierbarkeit

Skalierbare Rechnerarchitektur. Die steigende Zahl an angebotenen Modellen und deren Varianten[32] erfordert eine einfache Konfigurierbarkeit der Software im Fahrzeug. Die **Abbildung 2.8** zeigt in einer abstrakten Darstellung, wie der Echtzeitparallelrechner *ConPar* mittels vorgefertigter Software-Funktionsmodule je nach Ausstattung des Fahrzeugs konfiguriert werden könnte. Aus einem Pool von Softwaremodulen werden die benötigten Module als Paket zusammengestellt und in den Echtzeitparallelrechner geladen. Die notwendige Rechenleistung skaliert mit dem Umfang der Softwarepakete, was wiederum von der Fahrzeugausstattung abhängt. Aus diesem Grund muss *ConPar* eine skalierbare Rechnerarchitektur aufweisen, um entsprechende Rechenleistungen zur Verfügung zu stellen.

Portierbarkeit von Funktionssoftware. Die in der Abbildung 2.8 dargestellte Konfiguration unterschiedlicher Modelle erfordert die Portierbarkeit von modularer Funktionssoftware auf unterschiedliche Rechenhardware. Die Forderung nach Portierbarkeit ist eines der zentralen Ziele von AUTOSAR und wird mit dessen Einführung in der Steuergeräteentwicklung erfüllt[33].

[32] vgl. 2.2.4 Modularität und Skalierbarkeit, S. 16 f.

[33] vgl. 2.2.5 Portierbarkeit, S. 17 f.

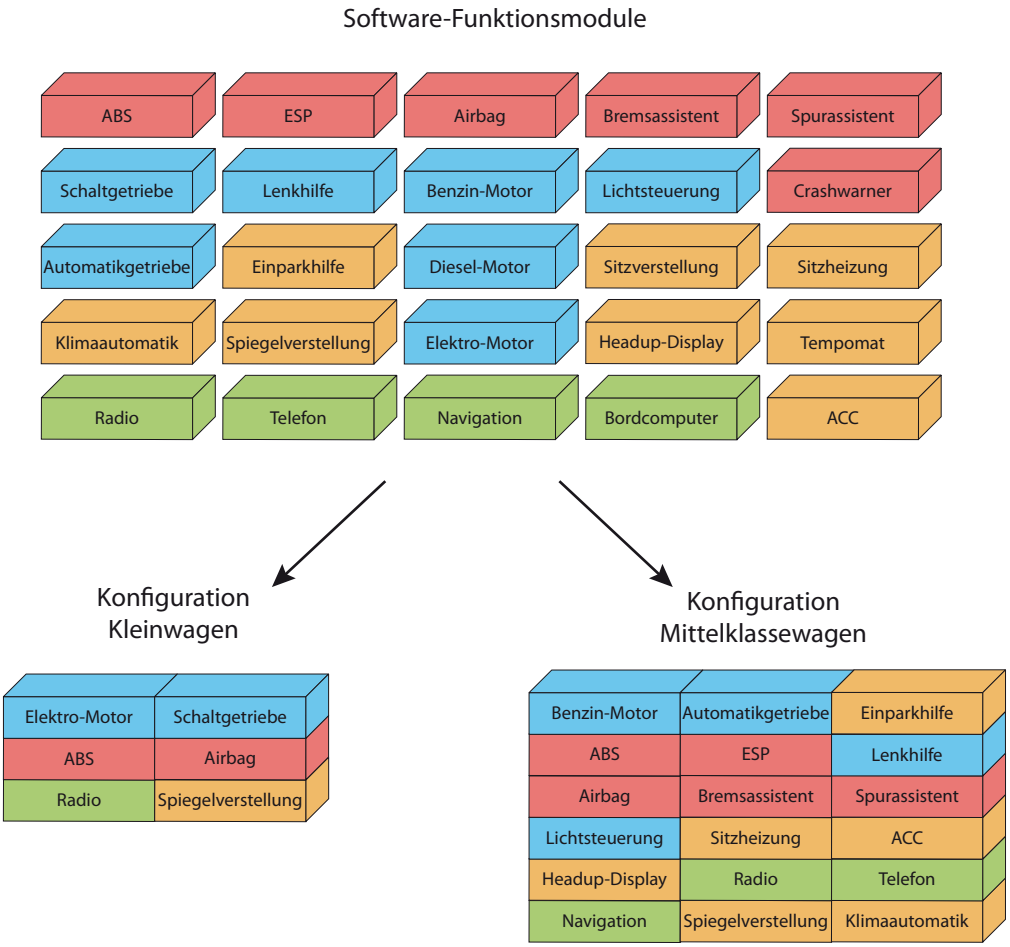


Abbildung 2.8: Konfiguration mittels Softwaremodulen

2.4 Fazit

Komplexe Bordnetzarchitektur. Die erlebbare Funktionalität moderner Fahrzeuge wird zunehmend durch Software bestimmt. Infolgedessen steigt nicht nur die Anzahl der Steuergeräte im Fahrzeug, sondern auch die Komplexität der Bordnetzarchitektur. Eine der großen Herausforderungen in der Zukunft besteht darin, die Entwicklungskosten von Fahrzeugen trotz des wachsenden Variantenreichtums und kürzerer Laufzeiten in Grenzen zu halten.

Vereinfachung durch Rezentralisierung. Einer der Hauptgründe für die Komplexität der bestehenden Bordnetzarchitektur besteht in der hohen Zahl der Steuergeräte, welche durch eine Rezentralisierung der Funktionssoftware mit einigen wenigen Steuergeräten deutlich gesenkt werden könnte. Gleichzeitig könnten die E/E-Architektur vereinfacht und weitere Herausforderungen, wie die Systemintegration oder die Portierbarkeit bestehender Software, bewältigt werden.

Kapitel 3 - Echtzeit- und Parallelrechner

Inhalt. Für die Rezentralisierung der Steuergeräte im Automobil nach dem Funktionsmodell von *ConPar* müssen große Mengen an zeitkritischen Daten verarbeitet werden. Mit dieser Forderung treffen zwei bisher eigenständige Rechnerwelten zusammen, welche sich unabhängig voneinander entwickelt haben: die *Echtzeitrechner* und die *Parallelrechner*. Aus diesem Grund soll das vorliegende Kapitel die Besonderheiten von Echtzeit- und Parallelrechnern herausstellen und zeigen, ob und wie beide Welten in einem *Echtzeitparallelrechner* vereint werden können.

3.1 Echtzeitrechner

Inhalt. Automobile Steuergeräte fallen aus rechentechnischer Sicht in die Gruppe der Echtzeitrechner. Echtzeit-Rechensysteme besitzen gegenüber konventionellen Rechensystemen einige besondere Eigenschaften, die in diesem Abschnitt näher betrachtet werden.

3.1.1 Grundlagen

Die Bedeutung von Echtzeit. In der Informatik bezieht sich der Begriff *Echtzeit*[1] auf das zeitlich korrekte Verhalten eines Rechensystems bei der Verarbeitung von Informationen. Die besondere Situation der Echtzeitdatenverarbeitung ist dadurch gekennzeichnet, dass das Rechensystem wie in der **Abbildung 3.1** dargestellt seine Eingabendaten unmittelbar aus seiner Umgebung bezieht und die Rechenergebnisse auf das umgebende System ebenso unmittelbar zurückwirken [Rz94, S. 13 ff.].

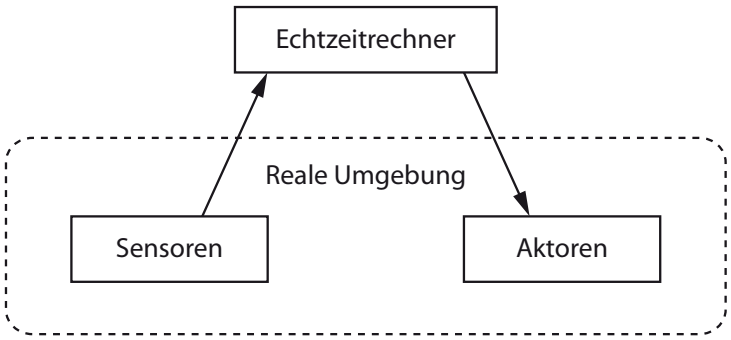


Abbildung 3.1: Komponenten eines Echtzeitsystems

[1] auch: Realzeit, abgeleitet vom englischen Begriff *real-time*

Aus diesem Grund muss das Rechensystem die Daten in der gleichen Geschwindigkeit bearbeiten wie sie auch anfallen. Das bedeutet, dass das Rechensystem mit der umgebenden (realen) Zeitbasis arbeiten muss. Im Unterschied zu einer Simulation, bei der zeitliche Abläufe beliebig gedehnt oder gerafft werden können, wird das Zeitverhalten von Echtzeitsystemen von der realen Umgebung bestimmt. Eine Definition für den Echtzeitbetrieb eines Rechensystems befindet sich in der DIN 44300[2]: „Ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen“ [Sc05a, S. 39].

Zeitlicher Determinismus. Aus der zeitlichen Abhängigkeit von der realen Umgebung folgt, dass bei der Datenverarbeitung in einem Echtzeitsystem neben der logischen Korrektheit der Rechenergebnisse auch deren zeitliche Korrektheit von entscheidender Bedeutung ist [Bu97, S. 4]. Ein Rechenergebnis behält seine Gültigkeit nur innerhalb festgelegter Zeitschranken (Deadlines). Außerhalb dieser Zeitschranken verliert das Rechenergebnis seinen Nutzwert. Man spricht von einem Echtzeitverhalten, wenn die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Dabei gilt jedoch nicht der Grundsatz der möglichst kurzen Antwortzeiten, da neben oberen Zeitschranken auch untere Zeitschranken zu einem Zeitfenster definiert werden können. Deshalb liegt die besondere Anforderung eines Echtzeit-Rechensystems darin, dass die Rechenergebnisse nicht so schnell wie möglich, sondern *so schnell wie nötig* zur Verfügung stehen. Daraus folgt, dass das Zeitverhalten des Echtzeit-Rechensystems innerhalb zulässiger Toleranzen vorherbestimmbar sein muss. Man spricht in diesem Zusammenhang von einem zeitlichen Determinismus des Rechensystems [Rz94, S. 14]. Eine solche Echtzeitanforderung findet sich zum Beispiel in der Steuerung eines Verbrennungsmotors in einem Fahrzeug. Die Zündung darf nicht zu spät erfolgen (obere Zeitschranke), aber auch nicht zu früh (untere Zeitschranke). Eine Reaktion außerhalb des Zeitfensters gilt als Fehlzündung und kann zu einer irreversiblen Schädigung des Motors führen. In Zahlen ausgedrückt, muss das Rechensystem bei einer maximalen Drehzahl von 6900 min^{-1} und einer Zündwinkeltoleranz von einem halben Grad in einem Zeitfenster von etwa 12 µs reagieren.

[2] inzwischen abgelöst durch die DIN ISO/IEC 2382

Harte und weiche Echtzeitanforderungen. Die unterschiedlichen Anforderungen an ein Echtzeit-Rechensystem werden durch ihren Härtegrad kategorisiert. Für die Beurteilung einer Echtzeitanforderung werden primär zwei Aspekte berücksichtigt: der zeitabhängige Nutzen des Rechenergebnisses und die Toleranz des Systems gegenüber der Verletzung von Zeitschranken.

Zeitabhängiger Nutzen des Ergebnisses. Zunächst wird die Frage gestellt, wie sehr die Verletzung der Zeitschranken den Nutzwert des Rechenergebnisses beeinflusst. Dazu lässt sich der (relative) Nutzen des Ergebnisses über einer Zeitachse auftragen (**Abbildung 3.2**). Bei einer harten Echtzeitanforderung wird das Ergebnis nach dem Erreichen der Zeitschranke sofort unbrauchbar. Bei einer weichen Echtzeitanforderung nimmt der Nutzwert dagegen kontinuierlich ab [Be09, S.3].

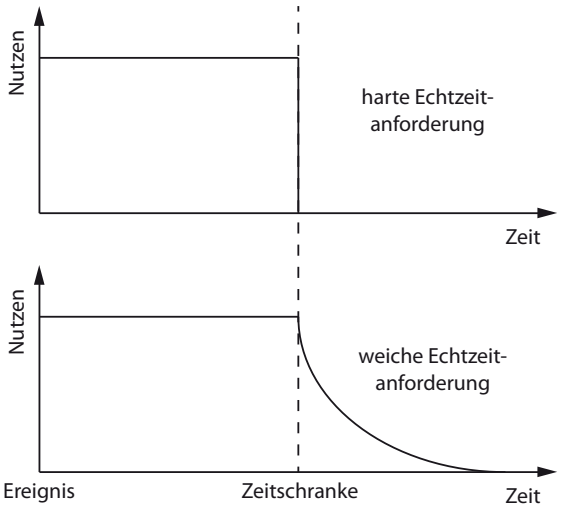


Abbildung 3.2: Harte und weiche Echtzeitanforderung

Toleranz gegenüber der Verletzung von Zeitschranken. Nun stellt sich die Frage, ob oder wie weit zeitliche Abweichungen toleriert werden können. Bei der Übertragung von Videosignalen in einer Rückfahrkamera zum Beispiel kann eine Verletzung der Zeitschranken zu einer verminderten Bildqualität (Bildflackern) führen. Darüber hinaus sind jedoch keine weiteren Konsequenzen zu erwarten. Es handelt sich hier um eine weiche Echtzeitanforderung, da das System zwar zeitempfindlich, aber nicht zeitkritisch ist. Dagegen kann die Verletzung der Zeitschranken bei einem aktiv eingreifenden System, wie etwa einer elektronischen Regelung der Fahrdynamik, Fehlfunktionen mit katastrophalen Konsequenzen für Mensch und Material verursachen. Hierbei handelt es sich um eine harte Echtzeitanforderung, bei der die Zeitschranken in jedem Fall einzuhalten sind [Sc05a, S. 40].

Periodische und aperiodische Ereignisse. Die besondere Aufgabe eines Echtzeit-Rechensystems besteht darin, auf Ereignisse aus der realen Umgebung innerhalb eines definierten Zeitfensters zu reagieren. Ereignisse werden aufgrund der besonderen zeitlichen Anforderungen in Echtzeitsystemen durch die Art ihres zeitlichen Auftretens charakterisiert. Die Unterscheidung erfolgt dabei in periodisch auftretende Ereignisse und aperiodisch auftretende Ereignisse. Periodische Ereignisse werden meist durch Taktgeber oder zum Beispiel in Fahrzeugen durch Drehzahlsensoren erzeugt. Solche Ereignisse sind durch ihre zeitliche Vorhersagbarkeit sehr gut planbar. Aperiodische Ereignisse können dagegen spontan auftreten und sind aus diesem Grund weitgehend nichtdeterministisch. Sie können lediglich aufgrund der Wahrscheinlichkeit ihres Auftretens (oder auch ihres Nichtauftretens bei gesicherten Pausenintervallen) berücksichtigt werden.

3.1.2 Datenverarbeitung in Echtzeit

Reaktionszeit. Damit die Echtzeitfähigkeit eines Rechensystems gesichert werden kann, muss die Reaktionszeit[3] des Rechensystems auf interne und externe Ereignisse bekannt sein. Eine wichtige Größe für die Bewertung der Echtzeitfähigkeit ist die längste mögliche Ausführungszeit WCET. Die WCET beschreibt die mögliche obere Zeitschranke des Systems. Daneben können, zum Beispiel bei Regelungsaufgaben, auch Schwankungen in der Ausführungszeit eine Rolle spielen. Aus diesem Grund ist neben der WCET auch die kürzeste mögliche Ausführungszeit BCET zu ermitteln. Der Quotient aus WCET und BCET zeigt die mögliche Schwankungsbreite (Jitter). Je enger beide Größen zusammenliegen, desto besser ist das System für Echtzeitanwendungen geeignet [Wö05, S. 152].

Ereignisgesteuerte Datenverarbeitung. Ein echtzeitfähiges Rechensystem muss in der Lage sein, periodisch wie auch aperiodisch anfallende Daten rechtzeitig zu verarbeiten. Aperiodisch auftretende Ereignisse, wie etwa der Signalwechsel an einem Sensor, werden meist ereignisgesteuert (*event triggered*) verarbeitet. Bei der ereignisgesteuerten Datenverarbeitung wird zum Zeitpunkt des Auftretens eines Ereignisses ein Interrupt ausgelöst. Das System kann somit ohne zusätzliche Latenz sofort auf das Ereignis reagieren, weshalb dieses Verfahren minimale Antwortzeiten bietet. Da aperiodische Ereignisse in den meisten Fällen zufällig auftreten und somit zeitlich nichtdeterministischen Charakter besitzen, ist die Echtzeitfä-

[3] Summe aus Latenz- und Servicezeit, vgl. Abbildung 3.3 auf Seite 29

higkeit ereignisgesteuerter Systeme aufgrund der schlechten zeitlichen Vorhersagbarkeit auf wenige Ereignisse beschränkt. Insbesondere dann, wenn mehrere Ereignisse gleichzeitig eintreten können, wird eine Ermittlung der WCET sehr schwierig. Wenn überhaupt, so kann die WCET nur aufgrund der Wahrscheinlichkeit des Eintretens mehrerer zeitgleicher Ereignisse vorherbestimmt werden. Ereignisgesteuerte Systeme eignen sich deshalb eher für die Verarbeitung von einigen wenigen und sehr schnellen Ereignissen.

Zeitgesteuerte Datenverarbeitung. Eine Alternative zur ereignisgesteuerten Datenverarbeitung bietet die zeitgesteuerte (*time triggered*) Datenverarbeitung. Zeitgesteuerte Systeme erfassen und verarbeiten Daten zu festgelegten Zeitpunkten[4]. Durch die Verwendung solcher Zeitraster kann die Ausführungszeit sehr genau geplant und Schwankungen minimiert werden. Demgegenüber steht die spekulative Allokation von Rechenleistung, auch wenn keine Reaktion nötig ist. Zudem ist die Latenzzeit abhängig vom Zeitpunkt des Ereignisses (vgl. **Abbildung 3.3**). Je kleiner das erlaubte Zeitfenster, desto enger wird das benötigte Abtastraster und desto höher wird die allokierte Rechenleistung. Darüber hinaus müssen sehr kurze Signale (Impulse) vom System gepuffert werden, damit Ereignisse nicht im Abtastraster verloren gehen (Shannon'sches Abtasttheorem[5]) [Sc05a, S. 41].

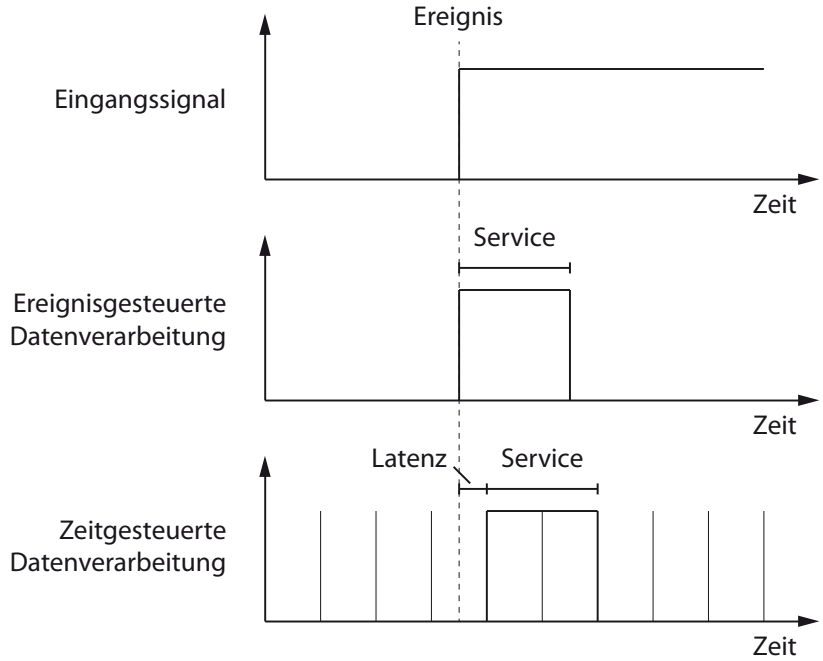


Abbildung 3.3: Ereignis- und zeitgesteuerte Datenverarbeitung

[4] z.B. eingesetzt bei der periodischen Abfrage von Sensoren (*sensor polling*)

[5] Claude Elwood Shannon (1916-2001), amerikanischer Mathematiker und Elektrotechniker

Für die praktische Anwendung in einer Steuerung empfiehlt sich nach meiner eigenen Erfahrung in vielen Fällen eine hybride Lösung aus beiden Verfahren, wobei der mehrheitliche Teil der Daten zeitgesteuert verarbeitet wird. Die unterbrechungsgesteuerte Verarbeitung beschränkt sich dagegen auf nur sehr wenige zeitkritische Ereignisse. Dadurch wird das zeitliche Verhalten des Rechensystems gut vorhersagbar, ohne die Reaktionsfähigkeit zu verlieren.

3.1.3 Herausforderungen für Echtzeitrechner

Rechtzeitigkeit. Neben der logischen Korrektheit der Rechenergebnisse ist deren rechtzeitige Verfügbarkeit innerhalb definierter Zeitschranken von entscheidender Bedeutung für die Echtzeitfähigkeit des Rechensystems. Außerhalb dieser Zeitschranken verliert das Rechenergebnis seinen Nutzwert und steht somit nicht mehr als Ergebnis zur Verfügung[6].

Nebenläufigkeit. Steuergeräte müssen multitaskingfähig sein, d.h. sie müssen mehrere Steuerungsaufgaben in Nebenläufigkeit ausführen können. Die zeitliche Koordinierung übernimmt ein Task-Scheduler, welcher die ausführbaren Tasks gemäß einer geeigneten Scheduling-Strategie zur Ausführung bringt. Hierfür existieren mehrere unterschiedliche Scheduling-Strategien[7], wobei in der praktischen Anwendung für Steuerungsaufgaben meist ein prioritätsbasiertes präemptives Scheduling genutzt wird. Hier können zeitkritische Aufgaben, welche durch hohe Prioritäten gekennzeichnet sind, andere Aufgaben jederzeit unterbrechen.

Vorhersagbarkeit. Vor allem bei sicherheitskritischen Systemen muss eine vorhersagbare Reaktion des Rechensystems in allen möglichen Situationen gewährleistet sein. Es darf zu keinem Zeitpunkt eine Situation eintreten, in der nicht klar ist, wie das System darauf reagieren wird. Diese Forderung ist insbesondere bei sehr komplexen Systemen mit vielen nebenläufigen Tasks nur schwer zu erfüllen.

Verfügbarkeit. Die rechtzeitige Verfügbarkeit der Rechenergebnisse muss in jedem Fall gewährleistet werden, d.h. auch bei einer Spitzenbelastung muss das Rechensystem noch rechtzeitig reagieren können. Dies gilt insbesondere für harte Echtzeitanforderungen, wo eine Verzögerung u.U. zu irreparablen Schäden[8] führt.

---

[6] vgl. 3.1.1 Grundlagen, S. 25 ff.

[7] z.B. Earliest Deadline First, Shortest Job First oder Round-Robin, vgl. [Co02, S. 17 ff.]

[8] vgl. 3.1.1 Grundlagen, S. 25 ff.

Fehlertoleranz. Ein Fehler in einer einzelnen Komponente darf nicht zu einem Ausfall des gesamten Systems führen. Diese Herausforderung gilt insbesondere für sicherheitskritische Systeme.

Wartbarkeit. Das System sollte so ausgeführt sein, dass sich Änderungen am Rechensystem nur auf den geänderten Teil auswirken, um den Aufwand einer erneuten Validierung des gesamten Systems zu vermeiden. Zudem sollte im Fehlerfall eine Analyse des Systems möglichst einfach durchzuführen sein.

3.1.4 Echtzeitfähige Hardware

Rechenleistung vs. Determinismus. Auf Seiten der Prozessor-Hardware existiert ein großes Portfolio an etablierten Techniken, um die Rechenleistung von Prozessoren effektiv zu steigern. Während bei einem normalen Rechensystem die Rechenleistung ganz oben auf der Agenda steht, ist bei einem Echtzeit-Rechensystem allerdings der zeitliche Determinismus das entscheidende Kriterium. Aus diesem Grund soll nachfolgend die Prozessorarchitektur auf ihre Echtzeitfähigkeit geprüft werden.

Caches. Da der Zugriff vom Prozessor auf den Hauptspeicher verhältnismäßig lange dauert, werden Caches als Puffer zwischen Prozessor und Hauptspeicher eingesetzt. Caches sind zwar sehr schnelle, aber auch ebenso kleine Speicher, so dass nicht alle möglichen Befehle oder Daten bevorratet werden können. Die Bevorratung erfolgt gemäß einer Strategie von Zugriffswahrscheinlichkeiten. Stehen die benötigten Befehle oder Daten nicht im Cache (*cache miss*), so kommt es aufgrund des langsameren Zugriffs auf den Hauptspeicher zu unplanbaren Verzögerungen.

Pipelining. In RISC-Prozessoren wird eine höhere Rechenleistung u.a. durch die Möglichkeit realisiert, dass Befehle in einer Pipeline wie am Fließband verarbeitet werden. Verzögerungen treten immer dann auf, wenn die Verarbeitung in der Pipeline an einer Stelle, z.B. aufgrund von fehlenden Operanden, ins Stocken gerät (*pipeline stall*). Diese Situationen sind aber i.d.R. vorsehbar und führen daher nicht zu einem indeterministischen Verhalten.

Sprungvorhersage. Die Befehlspipeline eines Prozessors wird, in Abhängigkeit von deren Länge, mit einem zeitlichen Vorlauf mit Befehlen gespeist. Bei einem bedingten Sprung im Programm kann eine falsche Sprungvorhersage zu einer Invalidierung der Pipeline (*pipeline flush*) führen. Die Sprungvorhersage folgt

ebenfalls einer Strategie von Sprungwahrscheinlichkeiten, was zu zeitlichen Schwankungen in der Programmabarbeitung (*service*) führt. Kürzere Pipelines führen hier zu weniger Verzögerungen bei falschen Sprungvorhersagen.

Dynamische und spekulative Befehlsausführung. Zur Vermeidung des Anhaltens der Pipeline können Befehle bei auftretenden Daten- oder Ressourcenkonflikten mittels dynamischer bzw. spekulativer Ausführung in der Pipeline vorgezogen oder zurückgestellt werden. Die zeitliche Vorhersagbarkeit des Rechensystems wird durch die dynamische Befehlsausführung nicht beeinträchtigt.

3.1.5 Echtzeitfähige Software

Multitasking. Die gleichzeitige Ausführung verschiedener Aufgaben führt in einem multitaskingfähigem Rechensystem zu einer Konkurrenzsituation zwischen lauffähigen Prozessen, da die verfügbare Rechenleistung adäquat an mehrere Prozesse verteilt werden muss. Da immer nur ein Prozess zu einem bestimmten Zeitpunkt vom Prozessor ausgeführt werden kann, erfolgt die Ausführung aller lauffähigen Prozesse lediglich „quasi-parallel“[9]. Jedem Prozess wird eine bestimmte Menge an Rechenleistung in Form von Zeitscheiben (*time slices*) zugewiesen. Die Zuweisung der Zeitscheiben an Prozesse übernimmt ein Task-Scheduler.

Task-Scheduling. Für Echtzeit-Rechensysteme besteht die besondere Forderung nach der garantierten Reaktion innerhalb vordefinierter Zeitschranken. Dafür bedarf es in einem Multitaskingbetrieb bestimmter Scheduling-Strategien, für die in der Vergangenheit eine Vielzahl unterschiedlicher Algorithmen und Methoden entwickelt wurden[10]. In der Vielzahl der existierenden Scheduling-Strategien gibt es allerdings nur ein entscheidendes Kriterium für die Echtzeitfähigkeit des Rechensystems. Um die rechtzeitige Reaktion auf Ereignisse zu garantieren, muss die Ausführung der laufenden Task unterbrechbar sein. Diese Forderung erfüllt ein präemptives Task-Scheduling. Bei einem kooperativen Task-Scheduling kann erst auf ein Ereignis reagiert werden, wenn die laufende Task den Prozessor freiwillig wieder an den Task-Scheduler frei gibt. Die **Abbildung 3.4** zeigt die unterschiedliche Reaktion beider Scheduling-Strategien bei Eintreten eines Ereignisses. Vor allem bei sicherheitskritischen Ereignissen ist ein kooperatives Scheduling aufgrund der unkalkulierbaren Wartezeit absolut undenkbar. In zweiter Reihe steht

[9] vgl. [Sc05a, S. 29]

[10] vgl. [He78], [Zi79] und [Sc05, S. 267 ff.]

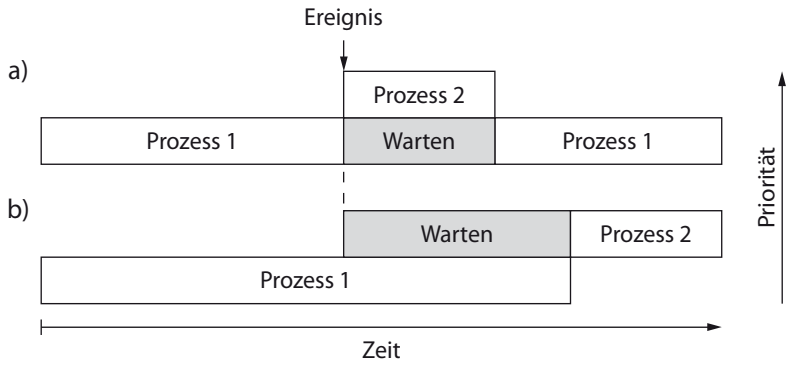


Abbildung 3.4: Task-Scheduling, a) präemptiv, b) kooperativ

die Unterscheidung von Scheduling-Strategien nach der Wahl der nächsten auszuführenden Task. Hier existieren bereits etablierte Methoden wie Earliest Deadline First, Shortest Job First, First-Come First-Served oder Round-Robin[11].

Prioritätsbasiertes Scheduling. In der Praxis wird für Steuerungsaufgaben aus Gründen der einfacheren Programmierung ein prioritätsbasiertes Scheduling mit statischen Prioritäten verwendet. Bei diesem Verfahren werden Zeitschranken durch Prioritäten ersetzt, d.h. es wird lediglich definiert, mit welcher Priorität eine Task ausgeführt werden soll. Das Kriterium für die Vergabe von Prioritäten durch den Programmierer liegt meist in der rechtzeitigen Beendigung der Task vor dem nächsten Ereignis. Die höchsten Prioritäten werden i.d.R. an zeitkritische und nur sehr kurz laufende Tasks vergeben, die innerhalb enger Zeitfenster reagieren müssen. Aufgrund der fehlenden Zeitschranken ist eine Analyse des zeitlichen Verhaltens zur Laufzeit ratsam.

Context Switching. Neben der Wahl einer geeigneten Scheduling-Strategie birgt der Multitasking-Betrieb von Echtzeitrechnern eine weitere Herausforderung durch das permanente Wechseln zwischen den Tasks (*context switching*). Jeder Taskwechsel beansprucht zusätzliche Rechenleistung in Form von Taskwechselzeit. In modernen Mikroprozessoren kann der Kontextwechsel bei Threads durch simultan mehrfädige Prozessoren (*simultaneous multithreading*[12]) in begrenztem Umfang beschleunigt werden[13]. Häufige Wechsel zwischen verschiedenen Tasks erhöhen aber trotzdem zwangsläufig die anteilige Rechenzeit, die für den Kontextwechsel zwischen den lauffähigen Tasks verwendet wird. In der Umkehrung erhöht sich dadurch die Zeit, in der andere lauffähige Tasks blockiert werden.

[11] vgl. [Si05, S. 218 ff.]

[12] z.B. bei der Hyper-Threading Technik in Intel-Prozessoren

[13] vgl. [Wö05, S. 151 f.]

3.1.6 Fazit

Der Zeitfaktor als wichtigstes Kriterium. In Echtzeitsystemen wird die erforderliche Rechengeschwindigkeit durch die Zeitbasis der realen Umgebung bestimmt. Entscheidendes Kriterium für Echtzeitrechner ist die Rechtzeitigkeit der Rechenergebnisse, die nur durch ein zeitlich deterministisches Verhalten sowohl bei der Hardware, als auch bei der Software erreicht werden kann.

Herausforderung durch Multitasking. Die für Echtzeitsysteme notwendige garantierte Einhaltung von Zeitschranken erzeugt aufgrund der permanent steigenden Anzahl von Aufgaben in automobilen Steuergeräten[14] eine immer größer werdende Komplexität. Neben der erforderlichen Rechenleistung stellt für konventionelle Echtzeitrechner vor allem das Task-Scheduling und die zeitliche Wechselwirkung verschiedener Tasks die größte Herausforderung dar.

3.2 Parallelrechner

Inhalt. Parallelrechner besitzen ihren Ursprung im Bereich des Supercomputings, weshalb ihre Architektur primär auf maximale Rechenleistung und nicht auf die Ausführung von Echtzeitaufgaben ausgerichtet ist. Allerdings besitzen Parallelrechner einige besondere konstruktive Eigenschaften wie Skalierbarkeit, Modularität und, wie der Name bereits impliziert, paralleler Programmausführung, welche bei der Rezentralisierung von Steuergeräten von ganz erheblicher strategischer Bedeutung sind und deshalb in diesem Abschnitt näher betrachtet werden sollen.

3.2.1 Rechnerarchitektur

„To pull a bigger wagon, it is easier to add more oxen than to grow a gigantic ox."
W. Gropp, E. Lusk, A. Skjellum [Gr96]

Was ist ein Parallelrechner? „Ein Parallelrechner ist eine Ansammlung von Berechnungseinheiten (Prozessoren), die durch koordinierte Zusammenarbeit große Probleme schnell lösen können" [Ra07, S. 17]. Abgesehen von Einzelfällen wie dem klassischen Beispiel der Wettervorhersage[15] als eine Echtzeitaufgabe im

[14] gilt auch für Steuerungen im Maschinen- und Anlagenbau

[15] die Berechnung der Wettervorhersage sollte mit einer gewissen Vorlaufzeit abgeschlossen sein, *bevor* das vorhergesagte Wetter eintritt

weitesten Sinne, spielt das Kriterium der Rechtzeitigkeit von Rechenergebnissen bei Parallelrechnern in deren Definition keine Rolle. Die Eigenschaft der schnellen Problemlösung, z.B. bei Computersimulationen, bezieht sich hier im Gegensatz zu Echtzeitrechnern auf *so schnell wie möglich*[16]. Trotz der eher unscharfen Formulierung sind in der oben genannten Definition jedoch die wesentlichen Architekturmerkmale eines Parallelrechners implizit enthalten. Zunächst existieren in einem Parallelrechner viele Prozessoren, die Befehle und Daten aus einem gemeinsamen oder verteilten Speicher vollständig parallel verarbeiten können. Damit die Prozessoren eine Aufgabe gemeinsam bearbeiten können, muss ein Kommunikationsmedium existieren, dass die Prozessoren zu einem Parallelrechner verbindet. Zudem benötigt ein Parallelrechner, wie alle anderen Rechensysteme auch, eine Schnittstelle (*interface*) zur Dateneingabe und -ausgabe. Daraus ergeben sich die folgenden vier wesentlichen Komponenten einer parallelen Rechnerarchitektur:

- Prozessoren,
- Speicher,
- Interprozessorkommunikation und
- externe Schnittstellen.

3.2.1.1 Prozessoren

Leistung durch Quantität. Parallelrechner erreichen ihre Rechenleistung primär durch eine sehr hohe Zahl an Prozessoren. So besitzt der derzeit weltweit schnellste Parallelrechner[17], der K Computer der Firma Fujitsu am RIKEN Advanced Institute for Computational Science (AICS) in Kobe (Japan), 68.544 Prozessoren vom Typ SPARC64 VIIIfx mit ingesamt 548.352 Rechenkernen. Die Summe aller Prozessorrechenleistungen führt zu der hohen Gesamtrechenleistung des Parallelrechners.

Einfache Prozessorarchitekur. Die Rechenleistung der einzelnen Prozessoren besitzt nur eine sekundäre Bedeutung für die erreichbare Gesamtrechenleistung, so dass die Architektur der Prozessoren in einem Parallelrechner auch sehr einfach ausgeführt sein kann. Auf diese Weise lassen sich relativ hohe Rechenleistungen auch mit sehr preisgünstiger Hardware realisieren[18].

[16] vgl. 3.1.1 Grundlagen, S. 25 ff.

[17] Stand: Juni 2011, www.top500.org

[18] vgl. Beowulf-Cluster [Ba06, S. 28 f.]

3.2.1.2 Speicher

Speicherorganisation. Ein wesentliches Element von Parallelrechnern stellt die Speicherorganisation dar. Die Speicherorganisation besitzt einen entscheidenden Einfluss auf das Programmiermodell des Parallelrechners. Bei Parallelrechnern erfolgt aus diesem Grund die Unterscheidung in zwei Arten von Rechensystemen:

- Rechensysteme mit verteilten Speicher und
- Rechensysteme mit gemeinsamen Speicher[19].

Verteilter Speicher. Bei einem Rechensystem mit einem verteilten Speichermodell (*distributed memory*) besitzt jeder Prozessor seinen eigenen Lokalspeicher (**Abbildung 3.5a**). Der direkte Zugriff auf den Speicher eines benachbarten Prozessors ist in diesem Speichermodell nicht vorgesehen.

Gemeinsamer Speicher. Bei einem Rechensystem mit gemeinsamen Speicher (*shared memory*) können alle Prozessoren auf den gesamten Speicherbereich direkt zugreifen. Dies geschieht entweder in einem virtuellen gemeinsamen Speicher, welcher durch einen DMA-Zugriff auf die Lokalspeicher benachbarter Prozessoren realisiert ist (**Abbildung 3.5b**) oder durch einen echten gemeinsamen Speicher in Form gemeinsamer Speichermodule (**Abbildung 3.5c**).

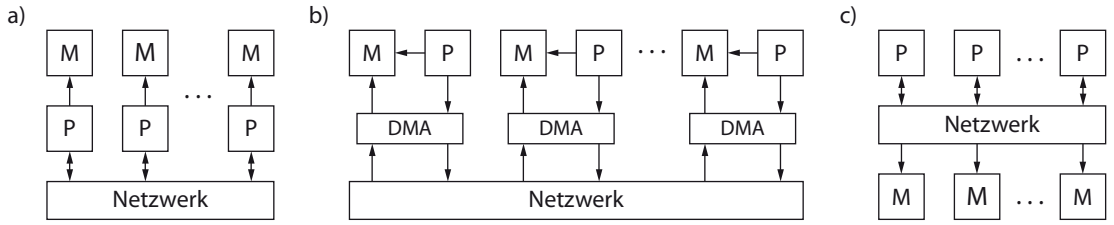


Abbildung 3.5: a) verteilter, b) virtueller gemeinsamer, und c) gemeinsamer Speicher[20]

3.2.1.3 Interprozessorkommunikation

Interprozessorkommunikation. Für die koordinierte Ausführung einer Rechenaufgabe ist eine geeignete Kommunikation zwischen den Prozessoren erforderlich. Die Interprozessorkommunikation kann im Parallelrechner, abhängig von dessen Rechnerarchitektur und Programmiermodell, durch das Senden und Empfangen

[19] vgl. [Ra07, S. 20 f.]

[20] P = Recheneinheit, M = Speichermodul

von Nachrichten (*message passing*) oder das Lesen und Schreiben von gemeinsamen Variablen (*shared variables*) erfolgen.

Austausch von Nachrichten. In einem Rechensystem mit verteiltem Speicher kann ein Prozessor lediglich auf seinen eigenen Lokalspeicher direkt zugreifen. Die Interprozessorkommunikation kann in diesem Fall nur über den Austausch von Nachrichten zwischen den Prozessoren realisiert werden.

Gemeinsame Variablen. In einem Rechensystem mit gemeinsamen Speicher können mehrere Prozessoren auf einen gemeinsamen Speicher zugreifen. Die Interprozessorkommunikation kann in diesem Fall über das kontrollierte Lesen und Schreiben von gemeinsamen Variablen realisiert werden.

Verbindungsnetzwerk. Die wesentliche Komponente eines Parallelrechners ist das Netzwerk, welche die Recheneinheiten miteinander verbindet und so die Interprozessorkommunikation ermöglicht. Mit Hilfe des Verbindungsnetzwerks werden Rechendaten und Steuerinformationen zwischen den Recheneinheiten vermittelt (vgl. **Abbildung 3.6**). Gerade bei sehr feingranularen Problemen, die auf sehr viele Recheneinheiten verteilt werden und häufig kommunizieren, ist das Verbindungsnetzwerk letztlich entscheidend für wichtige Eigenschaften des Parallelrechners wie Rechenleistung und Wirkungsgrad. Daneben besitzt die Topologie des Verbindungsnetzwerks durch verschiedene Metriken wie den Grad oder die kleinste Erweiterung[21] einen großen Einfluss auf die Skalierbarkeit des Parallelrechners.

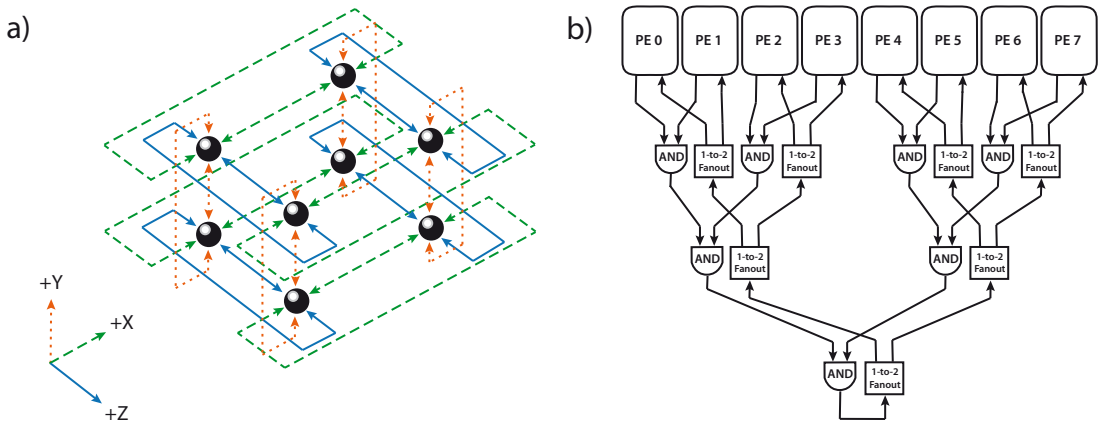


Abbildung 3.6: Kommunikation im Parallelrechner[22]: a) Datentransfertorus, b) Synchronisationsbaum

[21] vgl. *Metriken bei statischen Netzen* [Ri97, S. 83 ff.]

[22] grafische Darstellung aus [Cr93]

Anhand ihrer topologischen Eigenschaften lassen sich Verbindungsnetzwerke in zwei große Klassen einteilen: den *statischen* und den *dynamischen* Verbindungsnetzwerken[23], wobei in modernen Supercomputern häufig hybride Topologien eingesetzt werden. Während die Recheneinheiten i.d.R. aus Standardprozessoren bestehen, verwenden die Rechnerhersteller für das Verbindungsnetzwerk meist proprietäre Lösungen (vgl. **Tabelle 3.1**), was dessen strategische Bedeutung unterstreicht. So verwendete die Firma Cray in ihrem Modell T3E aus dem Jahr 1995 mit Baum und Torus zwei unterschiedliche statische Topologien für Synchronisation und Datentransport[24]. Ebenfalls im Jahr 1995 realisierte die Firma IBM in ihrem Modell SP2 eine dynamische Topologie mit Switch Chips[25]. Fast zwei Jahrzehnte später entwickelte die Firma Fujitsu für ihren K Computer eine hybride Topologie aus zwei 3-dimensionalen statischen Netzen, die zu einem 6-dimensionalen Netzwerk fusionieren[26].

Tabelle 3.1: Liste der weltweit zehn schnellsten Supercomputer[27]

Rank	Computer	Country	Manufacturer	Processor Technology	Processor	*Interconnect*
1	Sequoia	USA	IBM	PowerPC	Power BQC 16C 1.600GHz	*Custom*
2	K	Japan	Fujitsu	Sparc	SPARC64 VIIIfx 8C 2.000GHz	*Custom*
3	Mira	USA	IBM	PowerPC	Power BQC 16C 1.600GHz	*Custom*
4	SuperMUC	Germany	IBM	Intel SandyBridge	Xeon E5-2680 8C 2.700GHz	*Infiniband*
5	Tianhe-1A	China	NUDT	Intel Nehalem	Xeon X5670 6C 2.930GHz	*Proprietary*
6	Jaguar	USA	Cray Inc.	AMD x86_64	Opteron 6274 16C 2.200GHz	*Cray Gemini*
7	Fermi	Italy	IBM	PowerPC	Power BQC 16C 1.600GHz	*Custom*
8	JuQUEEN	Germany	IBM	PowerPC	Power BQC 16C 1.600GHz	*Custom*
9	Curie	France	Bull SA	Intel SandyBridge	Xeon E5-2680 8C 2.700GHz	*Infiniband*
10	Nebulae	China	Dawning	Intel Nehalem	Xeon X5650 6C 2.660GHz	*Infiniband*

[23] vgl. Kapitel 5 - Interprozessorkommunikation, S. 73 ff.

[24] vgl. *Das Verbindungsnetzwerk der Cray T3D/T3E* [Ri97, S. 252 ff.]

[25] vgl. *Das Verbindungsnetzwerk der IBM SP2* [Ri97, S. 266 ff.]

[26] vgl. *Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers* [Aj09]

[27] Stand: Juni 2012, www.top500.org

3.2.1.4 Externe Kommunikation

Externes Interface. Für einen konventionellen Parallelrechner dient die Kommunikation nach außen zur Eingabe von Rechenaufgaben und der Ausgabe von Rechenergebnissen. Die externe Kommunikation ist, im Gegensatz zur Interprozessorkommunikation, nicht maßgeblich für die Rechenleistung des Parallelrechners und spielt deshalb bei der Konstruktion von Parallelrechnern keine wesentliche Rolle.

3.2.2 Amdahl'sches Gesetz

Maximal erreichbare Rechenleistung. Das Amdahl'sche Gesetz[28] beschreibt die maximale Beschleunigung der Programmausführung, die durch den Einsatz von Parallelrechnern möglich ist. In der Literatur wird das Amdahl'sche Gesetz häufig kritisiert, da es viele Faktoren unberücksichtigt lässt, aber das soll an dieser Stelle nicht stören. Im Wesentlichen geht es bei Amdahl darum, dass der Leistungszuwachs auf den parallelisierbaren Anteil im Programm beschränkt ist. Die Gleichung 3.1 zeigt den nach Amdahl durch Parallelverarbeitung erreichbaren Speedup $S_P(N)$ mit P als parallelen Anteil im Programm und N als Anzahl der Prozessoren.

$$S_P(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{3.1}$$

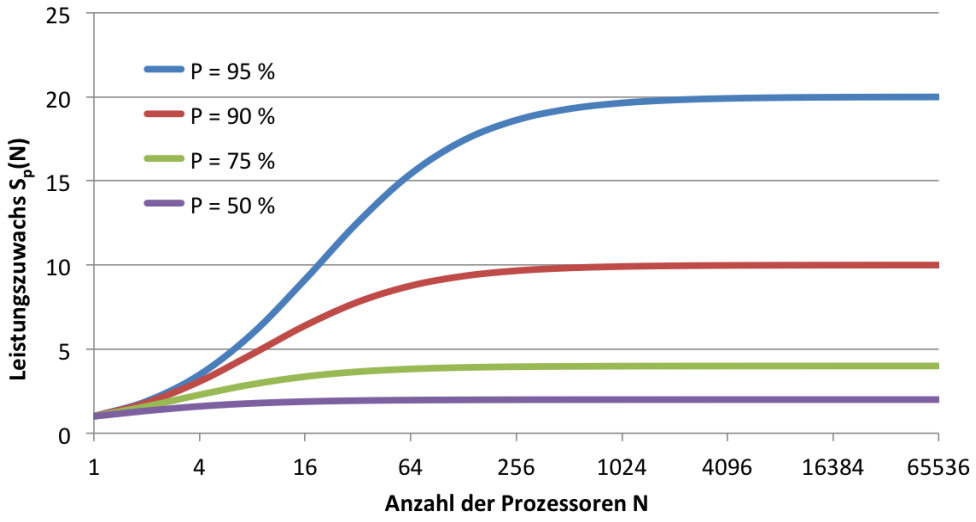


Abbildung 3.7: Leistungszuwachs durch Parallelverarbeitung nach Amdahl

[28] Gene Amdahl (geb. 1922), Computerarchitekt und Unternehmer

Sättigung durch sequentiellen Anteil am Programmcode. Der erreichbare Zuwachs hängt bei Amdahl maßgeblich vom Anteil an parallelisierbaren Programmcode ab[29] und kann nach Erreichen der Sättigung nicht durch den Einsatz von noch mehr Prozessoren gesteigert werden. So kann bei einem sequentiellen Anteil im Programmcode von 50 % die Rechenleistung durch Parallelverarbeitung maximal um den Faktor 2 gesteigert werden. Die **Abbildung 3.7** zeigt den Leistungszuwachs durch Erhöhung der Prozessorzahl gemäß dem Amdahl'schen Gesetz in Abhängigkeit vom Anteil an parallelisierbaren Programmcode.

3.2.3 Fazit

Hohe Rechenleistung. Parallelrechner bieten spezielle Architekturen zur Steigerung der Rechenleistung. Voraussetzung für den effizienten Einsatz von Parallelrechnern ist jedoch der Anteil nebenläufiger Strukturen im Anwenderprogramm, da sich die hohe Rechenleistung von Parallelrechnern prinzipbedingt aus der simultanen Verarbeitung einer Rechenaufgabe auf möglichst vielen Recheneinheiten ergibt. Die bestehende Software automobiler Steuergeräte müsste demnach in einer nebenläufig organisierten Form auf die Recheneinheiten im Parallelrechner abgebildet werden. Gelingt die Abbildung der bestehenden Steuergeräte-Software auf viele Prozessoren, so wäre es möglich, genügend Rechenleistung zu erzeugen, um mehrere Steuergeräte in einem einzigen Rechner zu rezentralisieren.

Skalierbarkeit. Neben der hohen Rechenleistung sind Parallelrechner auch durch skalierbare Rechnerarchitekturen charakterisiert. Das bedeutet, dass die Prozessorzahl und damit die erreichbare Gesamtrechenleistung je nach Umfang der auszuführenden Software im Fahrzeug angepasst werden kann, ohne dass wesentliche konstruktive Eigenschaften verändert werden müssen.

Verbindungsnetzwerk. Neben der Granularität der Rechenaufgabe stellt die Interprozessorkommunikation bzw. das Verbindungsnetzwerk ein wesentliches Kriterium dar. Die Bearbeitung von Aufgaben, die viel kommunizieren müssen, hängt neben den Recheneinheiten auch sehr stark von der Interprozessorkommunikation ab. Aus diesem Grund bestimmt das Verbindungsnetzwerk wesentliche Eigenschaften des Parallelrechners wie die Leistungsfähigkeit und die Skalierbarkeit bezüglich der Rechenleistung.

[29] vgl. [Ra07, S. 170 f.]

3.3 Die Architektur von *ConPar*

Inhalt. Der Echtzeitparallelrechner *ConPar*[30] vereint die wichtigsten Merkmale von Echtzeitrechnern[31] und Parallelrechnern[32] in einem einzigen Rechensystem und stellt somit einen Vertreter der neuen Klasse der *Echtzeitparallelrechner* dar. In diesem Abschnitt werden die besonderen Merkmale von Echtzeitparallelrechnern verdeutlicht, wobei das Hauptaugenmerk auf *ConPar* und der Rezentralisierung der Steuergerätesoftware liegen soll.

3.3.1 Besondere Eigenschaften

Echtzeitfähigkeit und Skalierbarkeit. Ein Echtzeitparallelrechner muss die besonderen Eigenschaften von Echtzeitrechnern als auch von Parallelrechnern in einem einzigen Rechensystem aufweisen. Das heißt im Wesentlichen, dass die Architektur des Echtzeitparallelrechners so gestaltet sein muss, dass das resultierende Rechensystem echtzeitfähig und skalierbar ist. Die **Abbildung 3.8** zeigt die wichtigsten besonderen Eigenschaften, die ein Echtzeitparallelrechner mitbringen muss.

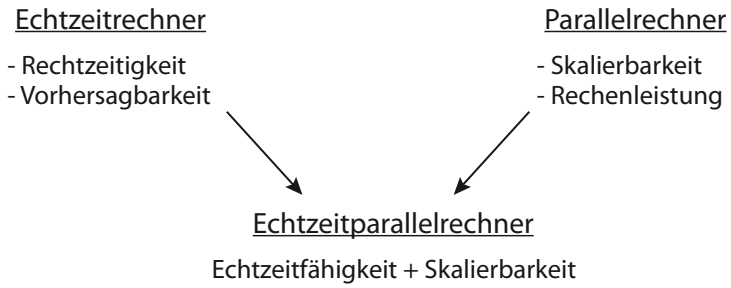


Abbildung 3.8: Besondere Eigenschaften eines Echtzeitparallelrechners

3.3.2 Klassifizierung

Multiple Instruction, Multiple Data. In der Flynn'schen Klassifizierung werden Parallelrechner anhand der auftretenden Daten- und Befehlsströme in vier charakteristische Klassen eingeteilt: SISD, SIMD, MISD und MIMD[33]. Für die Rezentralisierung automobiler Steuergeräte kommt nach dieser Einteilung für *ConPar* als

[30] Kunstbezeichnung aus den englischen Begriffen *control* und *parallel*

[31] vgl. 3.1 Echtzeitrechner, S. 25 ff.

[32] vgl. 3.2 Parallelrechner, S. 34 ff.

[33] vgl. [Ra06, S. 18 f.]

Echtzeitparallelrechner nur die Klasse der MIMD[34]-Rechner infrage, da in einem zentralen Steuergerät mehrere unterschiedliche Datenströme in der Form verschiedener Sensorsignale mit mehreren unterschiedlichen Befehlsströmen in der Form verschiedener Steuerungsprogramme verarbeitet werden müssen.

Art der Kopplung. Der entscheidende Faktor für das Verhalten und die Programmierung von parallelen Rechensystemen besteht in der Art der Kopplung der Prozessoren untereinander. Bei Parallelrechnern unterscheidet man zwischen eng gekoppelten Systemen[35], welche über einen gemeinsamen Speicher verfügen, und lose gekoppelten Systemen, die nur über Nachrichten kommunizieren[36]. Da die existierende E/E-Architektur im Fahrzeug aus einem verteilten System heterogener Steuergeräte besteht, dessen Kommunikationsmodell auf dem Austausch von Nachrichten basiert, kommt für *ConPar* ein lose gekoppeltes System infrage.

3.3.3 Parallele Datenverarbeitung

Mehr Rechenleistung durch Parallelisierung. Der Leistungszuwachs von Parallelrechnern beruht auf dem Ansatz, ein Problem fester Größe durch Parallelisierung zu lösen [Ba06, S. 16]. Das bedeutet, dass eine Aufgabe in kleinere Teilaufgaben zerlegt werden muss, welche dann von mehreren Recheneinheiten parallel bearbeitet werden können. Bei der Rezentralisierung der Software automobiler Steuergeräte entsteht so eine hierarchische Struktur zur parallelen Datenverarbeitung (**Abbildung 3.9**). Die oberste Ebene beinhaltet die einzelnen Steuergeräte. Hier findet in jedem Fall eine parallele Datenverarbeitung statt, da die Steuergeräte eines Fahrzeugs simultan arbeiten. Jedes einzelne Steuergerät beinhaltet

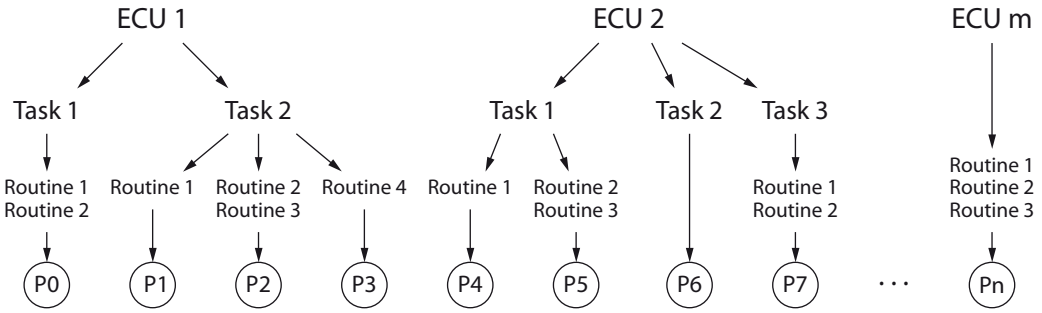


Abbildung 3.9: Aufteilung bestehender Steuergerätesoftware auf Recheneinheiten

[34] Abkürzung für Multiple Instruction, Multiple Data

[35] i.d.R. handelt es sich dabei Mehrkernprozessoren

[36] vgl. [Be09, S. 73]

wiederum eine Reihe von Tasks, die durch einen Task-Scheduler aufgerufen werden. Dies stellt die zweite Ebene dar, die ebenfalls ohne Einschränkungen parallel bearbeitet werden kann. Die dritte Ebene besteht aus einzelnen Routinen einer Task. Auf dieser Ebene muss im Einzelfall geschaut werden, ob zeitliche Abhängigkeiten zwischen zwei Routinen bestehen. Das in der Abbildung 3.9 dargestellte Beispiel zeigt, wie die hierarchische Aufteilung bestehender Steuergerätesoftware von m Steuergeräten auf n Recheneinheiten[37] in einem Parallelrechner erfolgen kann. Aus der hierarchischen Gliederung der rezentralisierten Steuergerätesoftware ergibt sich automatisch eine Abbildung paralleler Abschnitte auf Prozessoren, welche dem Amdahl'schen Gesetz[38] Rechnung trägt.

3.3.4 Rechnerarchitektur

Parallele Datenverarbeitung in Echtzeit. Die Architektur eines Echtzeitparallelrechners beruht aufgrund der parallelen Datenverarbeitung in mehreren Recheneinheiten auf der Architektur von konventionellen Parallelrechnern[39]. Im Unterschied zu konventionellen Parallelrechnern müssen die Komponenten des Echtzeitparallelrechners jedoch das vorhersagbare Verhalten von Echtzeitrechnern aufweisen, um eine Verarbeitung der Daten in Echtzeit zu gewährleisten.

3.3.4.1 Speichermodell

Verteiltes Speichermodell im Automobil. In einem modernen Automobil existiert ein heterogenes verteiltes System aus Steuergeräten, wobei aus der Sicht der Speicherarchitektur jedes Steuergerät eine in sich geschlossene Einheit bildet und somit einen Lokalspeicher darstellt. Die Kommunikation zwischen den Steuergeräten erfolgt ausschließlich durch den Austausch von Nachrichten über einen gemeinsamen Kommunikationsbus (**Abbildung 3.10**). Aufgrund der Heterogenität in der E/E-Architektur und der Kapselung der Steuergeräte nach außen ist der direkte Zugriff auf einen gemeinsamen Speicher[40] in der Automobilelektronik nicht realisiert und wird daher auch bei der Rezentralisierung bestehender Steuergerätesoftware im Echtzeitparallelrechner *ConPar* keine Rolle spielen.

[37] mit $m < n$

[38] vgl. 3.2.2 Amdahl'sches Gesetz, S. 39

[39] vgl. 3.2.1 Rechnerarchitektur, S. 34 f.

[40] vgl. Abbildung 3.5: a) verteilter, b) virtueller gemeinsamer, und c) gemeinsamer Speicher, S. 36

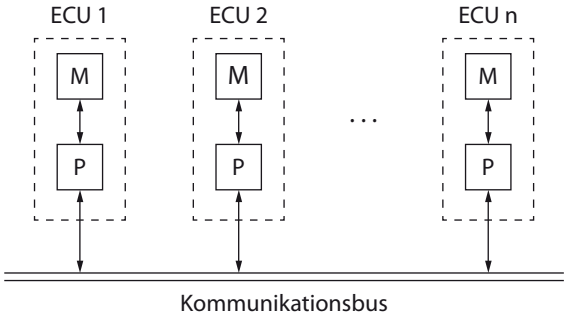


Abbildung 3.10: Speichermodell verteilter Steuergeräte (ECU) im Automobil[41]

3.3.4.2 Interprozessorkommunikation

Echtzeitfähigkeit. Parallelrechner erreichen ihre hohe Rechenleistung nur durch eine simultane Abarbeitung mehrerer nebenläufiger Teilaufgaben. Die Abarbeitung dieser Teilaufgaben erfolgt jedoch oft nicht unabhängig voneinander, weshalb bestimmte Recheneinheiten zu bestimmten Zeitpunkten auf die Rechenergebnisse anderer Recheneinheiten angewiesen sind. Im Echtzeitparallelrechner *ConPar* muss deshalb die Interprozessorkommunikation zwingend in Echtzeit erfolgen, da sonst die rechtzeitige Abarbeitung der Teilaufgaben nicht garantiert werden kann und damit die Echtzeitfähigkeit des gesamten Rechensystems nicht gegeben ist.

Latenz der Nachrichtenübertragung. Beim Echtzeitparallelrechner *ConPar* soll die Interprozessorkommunikation (analog zur Kommunikation zwischen Steuergeräten im Fahrzeug) auf dem Austausch von Nachrichten basieren. In der nachrichtenbasierten Kommunikation ist die Latenz der Nachrichtenübertragung ein wichtiger Parameter für die Echtzeitfähigkeit des Kommunikationsmediums. Nur bei einer vorhersagbaren Latenz der Nachrichten ist das Netzwerk echtzeitfähig.

Skalierbarkeit. Die Größe des Rechensystems darf für die maximale Latenz einer Nachricht keine Rolle spielen, da sonst zeitliche Grenzen verletzt werden könnten. So muss etwa das ABS in einem definierten Zeitfenster aktuelle Werte von den Radsensoren erhalten, unabhängig davon, ob zeitgleich die Motorsteuerung oder die Fahrwerksteuerung oder beliebig viele weitere Systeme parallel kommunizieren. Das bedeutet, dass das Verbindungsnetzwerk von *ConPar* hinreichend skalierbar sein muss, um die Interprozessorkommunikation unabhängig von der Größe des Rechensystems in Echtzeit realisieren zu können.

[41] P = Recheneinheit, M = Speichermodul

Echtzeitfähiges Verbindungsnetzwerk. Wie bei einem konventionellen Parallelrechner, so kommt auch beim Echtzeitparallelrechner dem Verbindungsnetzwerk eine herausragende Bedeutung zu, da es wesentliche Eigenschaften des Parallelrechners bestimmt. Aus diesem Grund ist in der vorliegenden Arbeit auch ein eigenes Kapitel der Interprozessorkommunikation von *ConPar* gewidmet. Das Kapitel 5 - Interprozessorkommunikation, S. 73 ff. beschäftigt sich mit der Frage, wie ein Netzwerk aussehen muss, um eine Interprozessorkommunikation in Echtzeit zu ermöglichen.

3.3.4.3 Externe Kommunikation

Ein- und Ausgabe von Echtzeitdaten. Im Unterschied zu einem konventionellen Parallelrechner[42] spielt beim Echtzeitparallelrechner der Informationsfluss von und nach außen eine ganz entscheidende Rolle. Bei Echtzeitrechnern müssen die Daten aus dem realen umgebenden System (im Fall von *ConPar* die Daten aus dem Fahrzeug) in Echtzeit, d.h. unter Einhaltung von Zeitschranken[43], eingelesen, verarbeitet und wieder ausgegeben werden. Aus diesem Grund übernimmt beim Echtzeitparallelrechner *ConPar* das externe Interface eine wichtige Schlüsselrolle.

3.3.5 Zusammenfassung

Parallele Datenverarbeitung. Das Konzept der vorliegenden Arbeit zur Rezentralisierung von Steuergeräten im Echtzeitparallelrechner *ConPar* beruht darauf, die gesamte Software im Fahrzeug in nebenläufige Teile zu organisieren und diese auf mehreren Recheneinheiten zeitlich parallel, d.h. in Nebenläufigkeit, zu verarbeiten. Insofern funktioniert der Echtzeitparallelrechner *ConPar* zunächst wie ein konventioneller Parallelrechner.

Kommunikation als Schlüsselfaktor. Der wesentliche Unterschied zwischen einem konventionellen Parallelrechner und dem Echtzeitparallelrechner *ConPar* liegt in der internen Kommunikation zwischen den Recheneinheiten sowie in der externen Kommunikation zwischen dem Rechensystem und dem umgebenden System. Nur wenn die gesamte Kommunikation in Echtzeit abgewickelt werden kann, ist das Rechensystem in der Lage, echtzeitfähig zu arbeiten.

---

[42] vgl. 3.2.1.4 Externe Kommunikation, S. 39

[43] vgl. *Die Bedeutung von Echtzeit* in 3.1.1 Grundlagen, S. 25 f.

Kapitel 4 - Space-Sharing

Inhalt. Der von mir vorgeschlagene Paradigmenwechsel vom Time-Sharing zum Space-Sharing in der Rechnerarchitektur bildet in der vorliegenden Arbeit die Grundlage für das Konzept zur Rezentralisierung von Steuergeräten im Automobil. Dieses Kapitel beschreibt die grundlegende Funktion von Space-Sharing und dessen Realisierung in einem FPGA-basierten Rechensystem. Es wird gezeigt, wie mit Hilfe von Space-Sharing ein Echtzeitparallelrechner[1] konstruiert werden kann, um mit einer skalierbaren Rechnerarchitektur sowohl einfache als auch sehr komplexe Applikationen in Echtzeit bewältigen zu können. Zudem werden einige wichtige Eigenschaften von Space-Sharing näher beleuchtet.

Ziele und Anforderungen. Das Ziel der vorliegenden Arbeit besteht darin, das verteilte System aus Steuergeräten im Automobil aus den im Kapitel 2 - Rezentralisierung von Steuergeräten, S. 9 ff. beschriebenen Gründen in einem Echtzeitparallelrechner zu rezentralisieren. Hierfür muss jedoch ein geeignetes Rechensystem entwickelt werden, welches in der Lage ist, die Aufgabe einer zentralen Steuerung zu übernehmen. Dabei gibt es eine ganze Reihe von Kriterien, die zu berücksichtigen sind. Das wichtigste Kriterium ist die Echtzeitfähigkeit des Rechensystems, um die Einhaltung von Zeitschranken im realen Fahrzeug zu garantieren. Vor allem bei sicherheitskritischen Anwendungen, wie z.B. ABS oder ESP, muss die Steuerung in jeder Situation rechtzeitig reagieren können[2]. Dabei dürfen auch Umfang und Komplexität der Steuerungsaufgabe bzw. die Anzahl zeitgleicher Aufgaben keine Rolle spielen. Als zweites wichtiges Kriterium muss die Architektur des Rechensystems die Forderung nach Skalierbarkeit erfüllen, um sowohl in Kleinstwagen als auch in Luxuslimousinen gleichermaßen eingesetzt werden zu können. Echtzeitfähigkeit und Skalierbarkeit lassen sich allerdings nur zusammenbringen, wenn nebenläufige Teile der Funktionssoftware parallel und unabhängig voneinander ausgeführt werden können.

4.1 Grundlagen

Inhalt. In diesem Abschnitt werden die Grundlagen zu Space-Sharing vermittelt. Dabei werden das Prinzip und die Umsetzung von Space-Sharing näher erläutert.

[1] vgl. 3.3 Die Architektur von ConPar, S. 41 ff.

[2] vgl. 3.1.3 Herausforderungen für Echtzeitrechner, S. 30 f.

4.1.1 Partitionierung von Rechenleistung

Scheduling von Multiprozessorsystemen. Der Begriff *Space-Sharing* wurde in der Vergangenheit bereits im Bereich des Schedulings von Multiprozessorsystemen verwendet. Die ursprüngliche Idee von Space-Sharing bestand darin, dass mehrere Threads eines Prozesses oft zusammenarbeiten und deshalb auch häufiger miteinander kommunizieren müssen. In solchen Fällen macht es Sinn, diese Threads zum gleichen Zeitpunkt auf mehreren CPUs parallel laufen zu lassen, wodurch eine direkte und zeitnahe Kommunikation ermöglicht wird. Dazu werden Threads ihrem Kommunikationsbedarf entsprechend in Gruppen zusammengefasst. Ein Scheduler überprüft zur Laufzeit, ob genügend CPUs für alle Threads einer Gruppe zur Verfügung stehen. Ist das der Fall, wird jedem Thread aus dieser Gruppe eine CPU zugewiesen. Sind nicht genügend CPUs verfügbar, dann wird solange kein Thread der Gruppe gestartet, bis genügend CPUs zur Verfügung stehen[3]. Das beschriebene Scheduling mehrerer Threads auf mehreren CPUs soll zur besseren Unterscheidung zukünftig als (*m*,*n*)-*Space-Sharing* bezeichnet werden, wobei die Anzahl *m* ausführbarer Threads größer ist als die Anzahl *n* verfügbarer CPUs.

Partitionierung der verfügbaren Rechenleistung. Das Prinzip des in [Ta09] beschriebenen (*m*,*n*)-Space-Sharings besteht darin, die Menge von *n* verfügbaren CPUs im Multiprozessorsystem in Partitionen einzuteilen. Die **Abbildung 4.1** zeigt in einem Beispiel die Partitionierung eines Multiprozessorsystems mit 32 CPUs in vier Partitionen unterschiedlicher Größe. Jeder Partition wird eine entsprechend große Gruppe aus Threads zugewiesen. In dem dargestellten Fall sind zwei CPUs

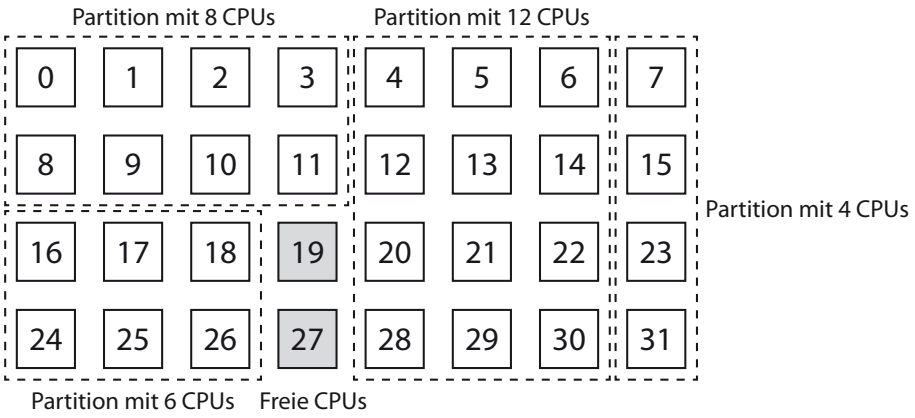


Abbildung 4.1: Partitionierung eines Multiprozessorsystems mit 32 CPUs[4]

[3] vgl. [Ta09, S. 636 f.]

[4] grafische Darstellung aus [Ta09, S. 637]

keiner Partition zugeordnet und können noch zwei einzelnen Threads oder einer Gruppe mit zwei Threads zugewiesen werden. Anzahl und Größe der Partitionen dürfen sich zur Laufzeit je nach Bedarf ändern. Sind genügend freie CPUs vorhanden, so wird eine neue Partition aus freien CPUs erstellt und einer Gruppe von Threads zugewiesen. Die Zuweisung einer CPU zu einem bestimmten Thread bleibt so lange gültig, bis dieser Thread terminiert. Erst danach wird die CPU für neue Aufgaben bzw. für die Erstellung einer neuen Partition freigegeben.

4.1.2 Space-Sharing für Echtzeitaufgaben

Neues Konzept von Space-Sharing. Nachfolgend soll der Begriff *Space-Sharing* in der von mir neu konzipierten Variante, dem (*n*,*n*)-Space-Sharing, verwendet werden. Das neue Konzept soll dazu dienen, eine skalierbare Rechnerarchitektur für Echtzeitaufgaben mit beliebig großer Komplexität zu schaffen[5]. Dieses Ziel kann nur erreicht werden, wenn das Task-Scheduling, welches auf konventionellen Echtzeitrechensystemen mit Multitasking-Betrieb bis dato die Achillesferse bei der Programmausführung darstellt, obsolet wird. Eine mögliche Lösung für dieses Problem soll mit dem neuen Konzept des (*n*,*n*)-Space-Sharings vorgestellt werden, wobei die Anzahl *n* verfügbarer CPUs der Anzahl *n* ausführbarer Tasks entspricht. Die Voraussetzung für Space-Sharing in Echtzeitsystemen bildet der Entwurf einer geeigneten Rechnerarchitektur[6] in Verbindung mit dem *Software-First-Design*[7], einem neuen Design-Paradigma für Echtzeitrechensysteme.

4.1.3 Zuordnung von Prozessoren

Von Multi-Core zu Many-Core. Die Grundidee von Space-Sharing, nebenläufige Teile einer Software zu partitionieren und anschließend parallel auf mehrere Prozessoren zu verteilen, soll auch in der neuen Variante von Space-Sharing erhalten bleiben. Allerdings ist die simultane Ausführung nun nicht mehr auf miteinander kommunizierende Tasks (bzw. Prozesse) beschränkt. Die Zahl der Prozessoren wird so weit erhöht, bis alle um die verfügbare Rechenzeit konkurrierenden Tasks gleichzeitig ausgeführt werden können, wodurch das bisher stets notwendige Task-Scheduling entfällt.

---

[5] vgl. 3.1.3 Herausforderungen für Echtzeitrechner, S. 30 f.

[6] vgl. 4.2 Rechnerarchitektur, S. 51 ff.

[7] vgl. 4.5.1 System-Design, S. 60 ff.

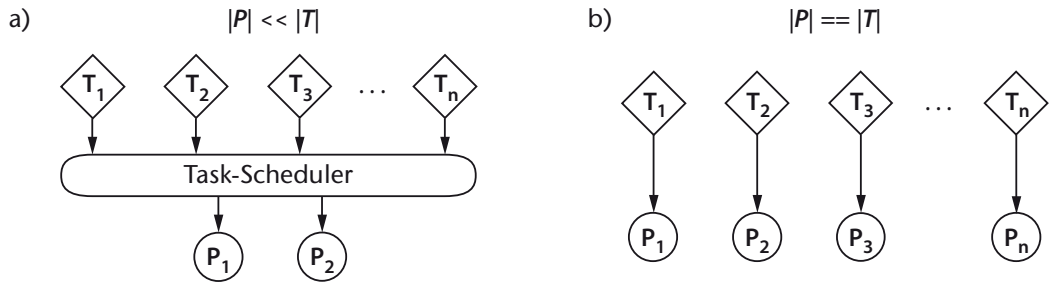


Abbildung 4.2: a) Time-Sharing, b) Space-Sharing

Time-Sharing vs. Space-Sharing. Der wesentliche Unterschied zwischen dem konventionellen Time-Sharing und dem neuen Space-Sharing ist in der **Abbildung 4.2** dargestellt. Im Time-Sharing entsteht eine Konkurrenz von *n* Tasks (*T*) um die Rechenleistung von wenigen (<< *n*) Prozessoren (*P*), welche durch einen Task-Scheduler gelöst werden muss (**Abbildung 4.2a**). Bei Space-Sharing wird die Konkurrenzsituation dadurch gelöst, dass für die Menge von *n* Tasks die gleiche Menge von *n* Prozessoren bereit steht (**Abbildung 4.2b**). Dafür ist, je nach Granularität und Komplexität der Anwendung, unter Umständen eine sehr hohe Zahl an Prozessoren notwendig, weshalb im Vergleich zu konventionellen Multi-Core-Rechnern (regulär mit 2 bis 8 Kernen) bereits von einem Many-Core-System gesprochen wird. In den nachfolgenden Abschnitten soll der Begriff *Multiprozessorsystem* gleichermaßen als Synonym sowohl für Multi-Core- als auch für Many-Core-Systeme verwendet werden.

Statische Zuordnung von Prozessoren. Eine Besonderheit von automobilen Steuergeräten, aber auch z.B. von industriellen Steuerungen, besteht darin, dass sich die Funktionssoftware und damit die Granularität der Anwendung zur Laufzeit nicht ändert. Obwohl bei einem Rechensystem prinzipiell Tasks (bzw. Prozesse) jederzeit erzeugt oder terminiert werden könnten, bleiben die Rechenaufgaben von automobilen Steuergeräten zur Laufzeit unverändert. Der Hintergrund besteht darin, dass sich das zu steuernde System, in diesem Fall das reale Fahrzeug, nicht ändert. Vor diesem Hintergrund kann die Funktionssoftware von Steuergeräten bereits zur Konfigurationszeit in nebenläufige Teile (Tasks bzw. Prozesse) partitioniert werden, welche den Prozessoren im Multiprozessorsystem statisch zur Ausführung zugewiesen werden. Die Zuweisung von Tasks auf Prozessoren muss zwingend statisch erfolgen, da sonst eine dynamische Lastverteilung im Multiprozessorsystem unter Echtzeitbedingungen nötig wäre, was jedoch bei sehr komplexen Aufgaben gegenüber dem Time-Sharing keinen Gewinn darstellt.

Deterministisches Verhalten trotz komplexer Aufgaben. Sind genügend Prozessoren vorhanden, d.h die Menge der Prozessoren (*P*) entspricht der Menge an Tasks (*T*), kann vom Prozessor-Multitasking abgesehen werden. Dadurch entfällt automatisch das beim Time-Sharing notwendige Task-Scheduling, welches insbesondere bei sehr komplexen Echtzeitaufgaben aufgrund von auftretenden Problemen, wie z.B. Prioritätsinversionen, schnell zu einem Gordischen Knoten mutiert, den es nun mittels Space-Sharing zu zerschlagen gilt. Durch den Wegfall des Task-Schedulings kann bei Space-Sharing das deterministische Verhalten des Rechensystems unabhängig von der Komplexität der Aufgabe sichergestellt werden. Das entscheidende Kriterium dafür ist die statische Abbildung von nebenläufigen Tasks (*T*) auf ebenso viele Prozessoren (*P*).

4.1.4 Besondere Eigenschaften

Skalierbare Rechenleistung. In einem Parallelrechner wächst die verfügbare Rechenleistung proportional mit der Anzahl der beteiligten Prozessoren. Der erreichbare Leistungszuwachs eines Parallelrechners ist allerdings abhängig vom parallelisierbaren Anteil in der Rechenaufgabe[8]. Da Space-Sharing die vorhandene Software in nebenläufige Teile partitioniert und statisch auf ebenso viele Prozessoren abbildet, skaliert die verfügbare Rechenleistung bei Space-Sharing mit der Anzahl der Prozessoren, was gleichzeitig dem Umfang der Software entspricht.

Räumliche Aufteilung der Software auf dem Chip. In einem Multiprozessorsystem on-Chip (MPSoC) okkupiert jeder Prozessor einen Teil der verfügbaren Fläche auf dem Halbleiter-Chip. Da jedem Prozessor im MPSoC wiederum ein Teil der Software zugewiesen wird, verdeutlicht der Begriff *Space-Sharing* im wörtlichen Sinne die räumliche Aufteilung der Software auf dem Chip [Au10a].

Vordefinierte Granularität der Rechenaufgabe. Eine wichtige Kenngröße beim Einsatz von Parallelrechnern ist die nebenläufige Granularität der Rechenaufgabe. Die zeitgleiche Abarbeitung einer Rechenaufgabe auf mehreren Prozessoren ist nur so weit sinnvoll, wie Teile der Aufgabe in Nebenläufigkeit ausführbar sind. Je feinkörniger die Aufgabe, desto mehr Prozessoren können an deren Umsetzung beteiligt werden. Ein signifikanter Unterschied zwischen automobilen Steuergeräten und konventionellen Parallelrechnern besteht darin, dass Art und Umfang der Software eines Steuergerätes und somit auch die Granularität der Aufgabe

[8] vgl. 3.2.2 Amdahl'sches Gesetz, S. 39 f.

schon vor der Laufzeit bekannt sind und sich zur Laufzeit auch nicht mehr ändern. Dadurch kann die Rechenlast bei Steuergeräten bereits zur Konfigurationszeit statisch auf mehrere Prozessoren verteilt werden. Eine dynamische Lastverteilung zur Laufzeit, wie sie im Allgemeinen in Parallelrechnern zum Einsatz kommt, wird bei Steuergeräten nicht benötigt [Au10b].

4.2 Rechnerarchitektur

Inhalt. Das Prinzip von Space-Sharing in seiner neuen Form besteht darin, beliebig viele nebenläufige Aufgaben auf ebenso viele Prozessoren statisch abzubilden. Dieses Prinzip bildet die Basis für ein Multiprozessorsystem, dessen Architektur in diesem Abschnitt beschrieben wird.

4.2.1 Space-Sharing mit FPGAs

Flexibilität von Hard- und Software. Die Echtzeitfähigkeit eines Rechensystems mittels Space-Sharing basiert darauf, dass genauso viele Recheneinheiten wie Tasks bzw. Prozesse existieren. Das bedeutet, dass beim Hinzufügen einer Task aufgrund einer Softwareänderung auch ein weiterer Prozessor in der Rechenhardware erforderlich wird. Dieses Ziel kann nur erreicht werden, wenn diese Hardware ähnlich flexibel wie Software angepasst werden kann. Diese Möglichkeit bieten FPGAs, bei denen die Rechnerarchitektur als Hardwarebeschreibung in VHDL erstellt, synthetisiert und zur Ausführung in den FPGA geladen werden kann. Zusätzlich zur Anzahl der Prozessoren können in VHDL weitere Parameter des Multiprozessorsystems, wie z.B. die Speichergröße oder verschiedene I/O-Verbindungen anwendungsspezifisch implementiert werden. Moderne FPGAs erlauben ein sehr komplexes MPSoC-Design mit einer Vielzahl an Prozessoren[9].

4.2.2 Aufbau und Komponenten

Multiprozessorsystem on-Chip (MPSoC). Die Realisierung von Space-Sharing erfolgt in einer parallelen Rechnerarchitektur, bestehend aus den vier Elementen *Prozessor*, *Speicher*, *Verbindungsnetzwerk* und *Ein-/Ausgabe-Interface*[10]. Jeder Prozessor bildet mit seinem lokal angeordneten Speicher eine modulare Rechen-

[9] vgl. 4.5.2 Realisierbarkeit komplexer MPSoCs mit FPGAs, S. 65 ff.

[10] vgl. 3.2.1 Rechnerarchitektur, S. 34 f.

einheit. Dadurch entsteht ein lose gekoppeltes System aus mehreren *Prozessor-Speicher-Modulen (PSM)*, welches als System-on-Chip (SoC) auf einem einzigen Chip implementiert wird (**Abbildung 4.3**). Die notwendige Interprozessorkommunikation wird durch ein Network-on-Chip (NoC) auf demselben Chip realisiert. Die Kommunikation nach außen erfolgt durch applikationsspezifische E/A-Module, welche je nach Bedarf an die einzelnen PSM oder für einen gemeinsamen Zugriff auch an das Netzwerk gebunden werden können.

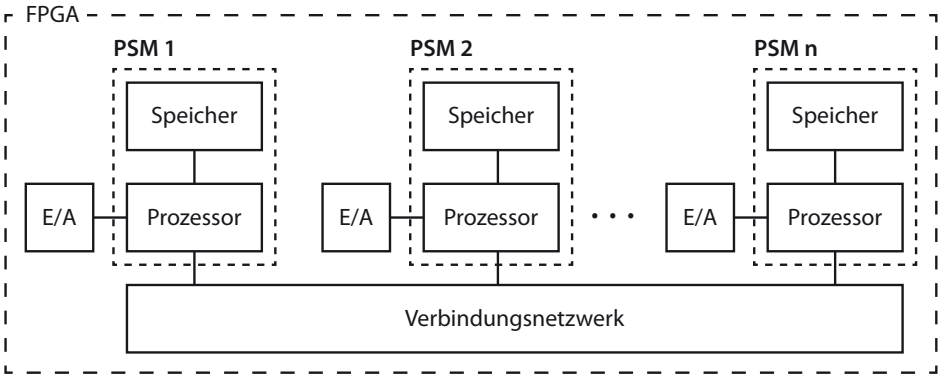


Abbildung 4.3: Lose gekoppeltes System aus Recheneinheiten (PSM)

Softprozessoren. Space-Sharing nutzt die Flexibilität von FPGA-Technologie, d.h. in den Recheneinheiten werden Softprozessoren verwendet. Softprozessoren existieren nicht als physikalische Hardware, sondern werden mittels einer Hardwarebeschreibungssprache wie VHDL in FPGA-Hardware implementiert, weshalb Softprozessoren in Verbindung mit einem *Software-First-Design*[11] sehr flexibel an die Bedingungen der auszuführenden Applikationssoftware angepasst werden können. So kann etwa der Befehlssatz des Softprozessors auf das Anwenderprogramm abgestimmt werden. Auf diese Weise lassen sich anwendungsspezifische Prozessoren entwerfen, wodurch die zur Verfügung stehenden Ressourcen im FPGA optimal genutzt werden können.

Koprozessoren. Die Rechenleistung der PSM kann durch den Einsatz von hardwarebeschleunigten Koprozessoren deutlich erhöht werden. Spezielle Funktionen, wie die FFT, können als Koprozessor komplett in Hardware ausgeführt werden. Dazu wird der Koprozessor in einer Hardwarebeschreibungssprache als IP-Core entworfen und über ein eigenes Interface (z.B. FSL[12]) an den zu unterstützenden Softprozessor gebunden und im FPGA implementiert.

[11] vgl. 4.5.1 System-Design, S. 60 ff.

[12] Fast Simplex Link der Fa. Xilinx [Xi10]

Lokalspeicher. Die Speichergröße der PSM lässt sich ebenso wie die Rechenleistung der Prozessoren flexibel an das Anwenderprogramm anpassen. FPGAs ermöglichen die bedarfsgerechte Zuweisung von Speicherressourcen an die jeweilige Task, sofern die benötigte Speichergröße bereits zur Konfigurationszeit bekannt ist. Darüber hinaus erlaubt die zur Verfügung stehende Harvard-Architektur inklusive getrennter Speicherbusverbindungen für Befehls- und Datenspeicher eine zusätzliche Optimierung des Speicherbedarfs für jedes PSM.

Netzwerk. Die Kommunikation zwischen den PSM erfolgt über ein Kommunikationsnetzwerk, welches als NoC realisiert ist. Die Anbindung an die Prozessoren erfolgt aus Gründen der Echtzeitfähigkeit über ein eigenes Interface.

Ein-/Ausgabe. Die periphere Kommunikation erfolgt durch anwendungsspezifische E/A-Module, die als IP-Core realisiert werden. Ein IP-Core kann einfache digitale Signale, allgemein verfügbare Busverbindungen wie I^2C und SPI oder auch proprietäre Kommunikationsprotokolle enthalten. Die Anbindung an den Softprozessor erfolgt über ein eigenes Interface (z.B. FSL) oder den Prozessorbus.

4.2.3 Kommunikationsmodell

Austausch von Nachrichten. Für die Interprozessorkommunikation haben sich bei Parallelrechnern zwei verschiedene Kommunikationsmodelle etabliert: der Austausch von *Nachrichten* und die Verwendung *gemeinsamer Variablen*[13]. Im Nachfolgenden konzentriert sich der Entwurf der Rechnerarchitektur auf den Austausch von Nachrichten mit dem Hintergrund, dass auch die busbasierte Kommunikation zwischen den derzeit existierenden Steuergeräten auf dem Senden und Empfangen von Nachrichten beruht und durch nachrichtenbasierte Kommunikation im Echtzeitparallelrechner *ConPar* sehr gut nachgebildet werden kann.

Gemeinsame Variable. Die Kommunikation über gemeinsame Variablen ist bei Space-Sharing ebenfalls realisierbar. Hierzu kann ein PSM durch ein Speichermodul (*shared memory*) substituiert werden, auf das andere PSM gemeinsam schreibend und lesend zugreifen können. Der Datenzugriff erfolgt wiederum über das Verbindungsnetzwerk durch den Austausch von Nachrichten auf Basis einer Client-Server-Kommunikation, wobei das Speichermodul den Schreib-/Lesezugriff auf gemeinsame Daten als Dienst bereitstellt.

[13] vgl. 3.2.1.3 Interprozessorkommunikation, S. 36 f.

4.3 Zeitliche Isolation

Inhalt. Eine der größten Herausforderungen bei der Programmierung komplexer Echtzeitsysteme stellt der Multitasking-Betrieb des Rechensystems unter harten Echtzeitbedingungen dar[14]. Space-Sharing bietet hierfür nun die Möglichkeit, die Ausführung einer Task zeitlich zu isolieren und damit jede Task unabhängig von der gleichzeitigen Ausführung anderer Tasks zu machen. Im Unterschied zum Time-Sharing beschränken sich die Auswirkungen von Änderungen in der Laufzeit einer Task nur auf die Task selbst, während das Ausführungsverhalten anderer Tasks unberührt bleibt.

4.3.1 Herausforderungen bei Time-Sharing

Gleichzeitigkeit und Konkurrenz. In einem konventionellen Rechensystem werden mehrere Tasks zeitgleich im Multitasking ausgeführt. Solange die Zahl der auszuführenden Tasks die Zahl der verfügbaren Prozessoren übersteigt, erfolgt die Ausführung jedoch nur quasi-parallel, da sich mindestens zwei Tasks die Rechenleistung eines Prozessors teilen müssen. Dadurch entsteht eine Konkurrenzsituation der Tasks um die Rechenleistung des Prozessors als elementare Ressource, welche durch eine geeignete Scheduling-Strategie gelöst werden muss.

Unterschiedliche Startbedingungen. Damit eine Task ausgeführt wird, muss sie vom Task-Scheduler des Prozessors zur Ausführung aufgerufen werden. Steuergeräte besitzen grundsätzlich drei Möglichkeiten zum Aufruf einer Task, die in der **Abbildung 4.4** dargestellt sind. Es handelt sich dabei um eine kontinuierliche, zeitgesteuerte oder ereignisgesteuerte Ausführung. Die *kontinuierliche Ausführung* einer Task erfolgt über die gesamte Laufzeit des Steuergerätes hinweg. Nach dem vollständigen Durchlauf des Programms wird die Ausführung der Task ohne Unterbrechung am Beginn des Programms fortgesetzt. Die kontinuierliche Ausführung wird bei Multitaskingbetrieb genutzt, um Aufgaben ohne besondere Priorität im Hintergrund zu erledigen. Aufgrund der fehlenden Prioritäten dürfen diese Aufgaben keine Zeitbedingungen enthalten, also keine Echtzeitaufgaben sein. Neben der kontinuierlichen Ausführung gibt es die Möglichkeit, den Start der Ausführung einer Task an das Auftreten von zeit- oder ereignisgesteuerten Startbedingungen (*trigger*) zu binden. Bei einem zeitgesteuerten Aufruf wird

[14] vgl. 3.1.5 Echtzeitfähige Software, S. 32 f.

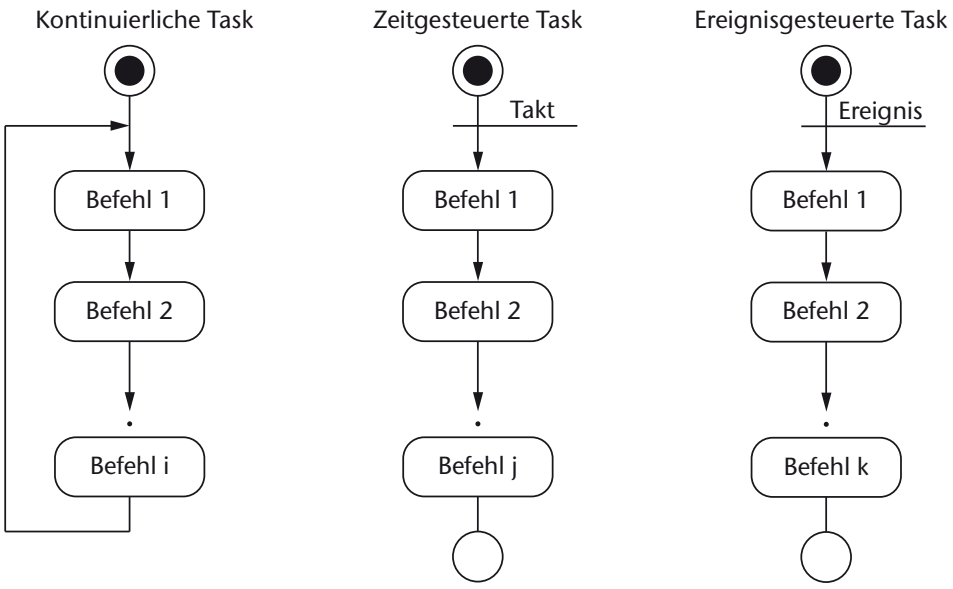


Abbildung 4.4: Zeitlich verschiedene Ausführungen von Tasks [Au10a]

die Ausführung der Task durch einen Taktgeber vom Rechensystem periodisch in festen Zeitintervallen gestartet. Die *zeitgesteuerte Ausführung* wird i.d.R. für periodisch zu wiederholende Aufgaben (z.B. beim Sensor-Polling) genutzt. Durch die Definition fester Zeitpunkte ist die Ausführung zeitgesteuerter Tasks sehr gut planbar. Sowohl hinsichtlich der Reaktionszeiten als auch hinsichtlich der Scheduling-Strategie bieten sie somit ideale Eigenschaften für ein zeitlich deterministisches Rechensystem. *Ereignisgesteuerte Aufrufe* können dagegen aperiodisch, d.h. spontan und zeitlich zufällig verteilt, auftreten. Ein solches Ereignis kann z.B. durch einen Sensor oder durch eine eintreffende Nachricht aus dem Netzwerk ausgelöst werden. In der Regel werden ereignisgesteuerte Aufrufe für Aufgaben mit sehr kurzen Reaktionszeiten bzw. sehr kleinem Jitter verwendet, weshalb ihnen auch die höchsten Prioritäten zugewiesen werden.

Variierende Ausführungszeiten (Jitter). Aufgrund des zeitlich zufällig verteilten Auftretens von Ereignissen und dem damit verbundenen Aufruf hochpriorer Tasks kommt es bei einer prioritätsbasierten Scheduling-Strategie zu Unterbrechungen in der Task-Ausführung durch zeit- oder ereignisgesteuerte Tasks höherer Priorität. Dadurch verlängert sich die Ausführungszeit der unterbrochenen Task, wodurch es, wie in der **Abbildung 4.5** dargestellt, zu längeren Reaktionszeiten kommen kann. Im Extremfall kann es sogar zu einer Überlappung von aufeinanderfolgenden Aufrufen einer Task kommen, wenn eine Task vor ihrem nächsten Aufruf nicht vollständig abgearbeitet wurde.

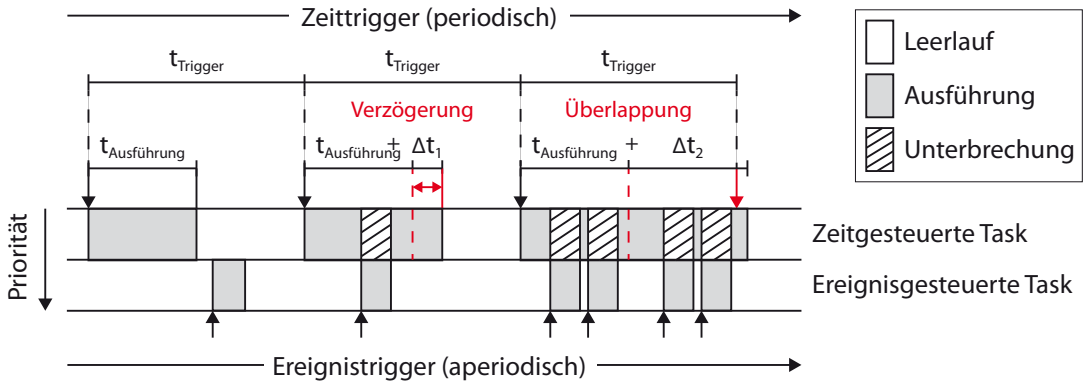


Abbildung 4.5: Verzögerung und Überlappung zur Laufzeit [Au10a]

Prioritäten. Eine weitere Herausforderung der prioritätsbasierten Scheduling-Strategie besteht in der Vergabe von Task-Prioritäten. An dieser Stelle muss der Programmierer der Anwendung entscheiden, welche Task mit welcher Priorität ausgeführt werden soll. Gerade bei sehr komplexen Anwendungen kann eine solche Entscheidung oder eine spätere Änderung gravierende Auswirkungen für die zeitliche Ausführung der Software auf dem Rechensystem besitzen.

Effekte bei Änderungen. Häufig müssen einzelne Tasks hinsichtlich ihres funktionalen Verhaltens nachträglich angepasst werden. Eine solche Änderung kann darin bestehen, dass Programmcode hinzugefügt wird, wodurch sich die Laufzeit der betreffenden Task verlängert. Dies kann wiederum zur Folge haben, dass eine niederpriore Task länger unterbrochen wird, weshalb sich auch deren Laufzeit verlängert. Auf diese Weise kann es im Worst-Case-Szenario dazu kommen, dass infolge einer kleinen Programmänderung das System teilweise oder vollständig funktionsunfähig wird. Einen ähnlichen Effekt hätten die Erhöhung der Frequenz von zeitgesteuerten Aufrufen oder das Hinzufügen einer neuen Task. Auch die Änderung von Prioritäten kann im schlechtesten Fall zu einem komplett anderen Zeitverhalten führen. Aus den genannten Gründen ist es bei Echtzeitsystemen unumgänglich, das zeitliche Verhalten des Systems nach Änderungen an der Software mit geeigneten Methoden zu prüfen.

4.3.2 Zeitliche Isolation durch Space-Sharing

Unterbrechungsfreie Ausführung. Bei Space-Sharing werden alle Tasks auf je ein eigenes Prozessor-Speicher-Modul (PSM) verteilt, wodurch das klassische Prozessor-Multitasking entfällt. Sobald eine Task ausführungsbereit ist, kann

diese ohne Verzögerungen oder Unterbrechungen abgearbeitet werden, da der ausführende Prozessor der ihm zugeordneten Task exklusiv zur Verfügung steht. Eine Scheduling-Strategie zur Ausführung aller Tasks ist damit obsolet. Durch die Ausführung der Tasks auf je einem eigenen PSM werden diese innerhalb des Rechensystems zeitlich voneinander isoliert.

Immunität vor Änderungen fremder Tasks. Änderungen am Programmcode oder am Scheduling einer einzelnen Task können sich bei konventionellen Rechensystemen mit Time-Sharing aufgrund der Wechselwirkung zwischen den Tasks auch auf das Laufzeitverhalten anderer Applikationen auf dem Rechensystem auswirken. Dies hätte wiederum Auswirkungen auf das Verhalten des gesamten Systems, also auch auf die Teile, bei denen keine Änderungen vorgenommen wurden. Bei Space-Sharing beschränkt sich die Auswirkung einer Änderung dagegen auf das PSM, dessen Programmcode geändert wurde. Die Ausführung von Applikationen auf einem anderen PSM bleibt vor dieser Änderung immun. Diese Eigenschaft ist insbesondere bei sicherheitskritischen Systemen von Bedeutung, da sich der Aufwand für eine erneute Validierung der korrekten Funktion des Systems auf die Software des geänderten PSM beschränkt.

Variation und Skalierbarkeit der Software. In der modernen Automobilelektronik besitzt das Thema der Skalierbarkeit eine herausragende Bedeutung. Aufgrund der dramatisch gewachsenen Palette an unterschiedlichen Modellen und Ausstattungen muss die Echtzeitfähigkeit des Rechensystems auch bei allen angebotenen Varianten erhalten bleiben. Das Hinzufügen einer einzigen Task aufgrund z.B. einer geänderten Ausstattung kann im Multitasking-Betrieb auf konventionellen Rechensystemen aufgrund der notwendigen Änderung im Task-Scheduling im Worst-Case zu einem komplett veränderten Systemverhalten führen. Bei Space-Sharing hingegen können beliebig viele Tasks (bzw. PSM) hinzugefügt werden, ohne dass bereits bestehende Tasks davon tangiert werden. Den limitierenden Faktor bei Space-Sharing bildet lediglich die Größe des Chips, auf dem die Software ausgeführt wird. Allerdings hat die Integrationsdichte der für Space-Sharing verwendeten FPGAs in den letzten Jahren sehr große Fortschritte erfahren. Extrapoliert man die Zahlen von Xilinx Spartan-3 und Virtex-4 FPGAs auf aktuelle FPGAs der Virtex-7 Generation, so können auf den heute verfügbaren Chips (theoretisch) mehrere hundert PSM implementiert werden[15].

[15] vgl. 4.5.2 Realisierbarkeit komplexer MPSoCs mit FPGAs, S. 65 ff.

Optimierung der Leistungsaufnahme. Im Gegensatz zu Time-Sharing kann bei Space-Sharing aufgrund der zeitlichen Isolation der Tasks auf verschiedenen PSM die Ausführungsgeschwindigkeit jeder Task unabhängig von der Ausführung anderer Tasks verändert werden. Die Taktrate jedes einzelnen PSM im Multiprozessorsystem kann dadurch im Rahmen eines GALS-Designs[16] dynamisch zur Laufzeit an die erforderliche Rechenleistung für die jeweilige Task angepasst werden[17], wodurch die dynamische Verlustleistung und damit der Energieverbrauch des Rechensystems positiv beeinflusst werden kann[18].

4.4 Räumliche Isolation

Inhalt. In einem konventionellen Rechensystem werden nebenläufige Tasks in einem gemeinsamen physischen Speicher ausgeführt. Das Betriebssystem hat im Multitasking-Betrieb die Aufgabe, mit Hilfe eines Speichermanagements virtuellen Speicher für alle Prozesse (bzw. Tasks) bereitzustellen und den Schutz des Speichers vor unberechtigtem Zugriff fremder Prozesse zu gewährleisten. Space-Sharing bietet aufgrund der Rechnerarchiktektur nun die Möglichkeit, Speicherbereiche auf dem Chip räumlich voneinander zu isolieren und somit automatisch vor unberechtigten Schreibzugriffen fremder Prozesse zu schützen.

4.4.1 Herausforderungen bei gemeinsamem Speicher

Speicherschutz durch Speichermanagement. In einem konventionellen Ein- oder Mehrprozessorsystem existiert i.d.R. ein gemeinsamer physischer Speicher, auf den alle Prozesse zugreifen. Der physische Speicher ist in logische Adressbereiche aufgeteilt, wobei jedem Prozess sein eigener virtueller Speicher zugewiesen wird (**Abbildung 4.6**). Der zugewiesene Speicherbereich beginnt bei einer bestimmten Basisadresse und besitzt eine definierte Größe. Innerhalb seines Bereiches hat der jeweilige Prozess alle Schreib- und Leserechte. Das Betriebssystem hat die Aufgabe, den Speicherbereich jedes Prozesses vor unberechtigten Zugriffen anderer Prozesse zu schützen. Der Schutz des Speichers ist Teil eines umfangreichen Speichermanagements und hat eine elementare Bedeutung für die Betriebssicherheit konventioneller Rechensysteme, da z.B. falsch programmierte Zeiger in einem

[16] vgl. 6.3.2 GALS-Design, S. 190

[17] vgl. 6.3.3 Dynamische Anpassung der Prozessorleistung, S. 190 ff.

[18] vgl. 6.2.1 Einfluss der Taktfrequenz, S. 178 f.

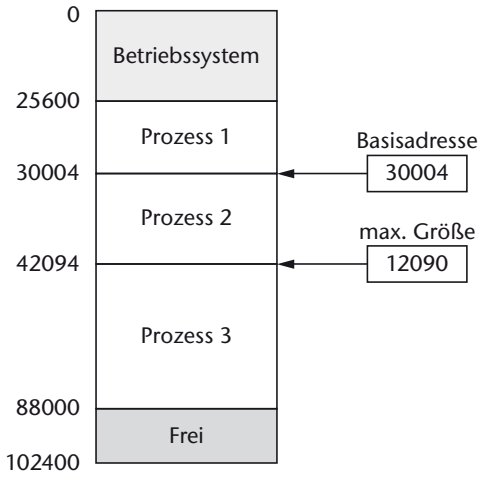


Abbildung 4.6: Aufteilung des physischen Speichers auf mehrere Prozesse[19]

Anwenderprogramm Daten oder Befehle fremder Prozesse unabsichtlich verändern und dadurch Fehlfunktionen verursachen könnten. Speicherschutzmechanismen sind deshalb bei automobilen Steuergeräten von herausragender Bedeutung für die Gewährleistung der sicheren Funktion [Eb07].

4.4.2 Räumliche Isolation durch Space-Sharing

Speicherschutz durch Lokalspeicher. Bei Space-Sharing wird jeder Prozessor mit einem eigenen Lokalspeicher ausgestattet. Die **Abbildung 4.7** zeigt das Beispiel aus der Abbildung 4.6 mit der Aufteilung der Prozesse auf eigene Lokalspeicher. Der Zugriff auf die Lokalspeicher erfolgt durch eine direkte und exklusive Speicherbusverbindung zwischen Prozessor und Speicher. Auf diese Weise erfolgt eine reale Trennung des Speichers auf physischer Ebene, so dass kein Prozess einen direkten Zugriff auf den Lokalspeicher eines anderen Prozesses in einem

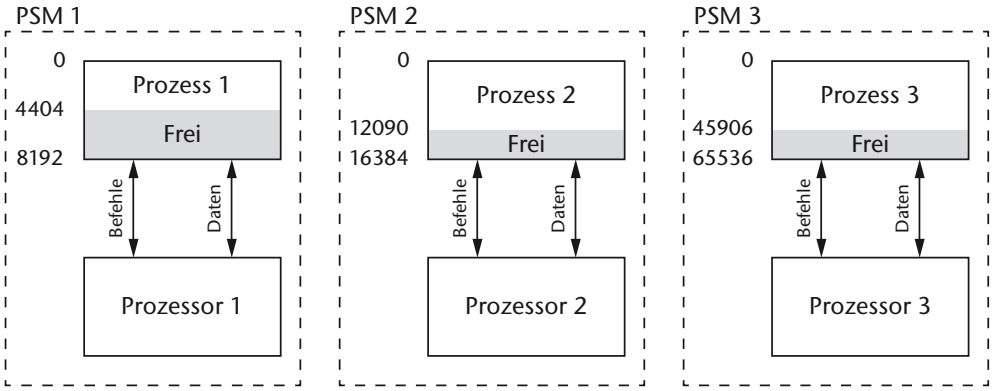


Abbildung 4.7: Aufteilung von Prozessen auf PSM

[19] grafische Darstellung nach [Si05, S. 276]

anderen PSM erlangen kann[20]. Die Aufteilung des gemeinsamen Speichers in virtuelle Speicherbereiche sowie der Schutz der virtuellen Speicherbereiche vor unberechtigten Zugriffen durch ein Speichermanagement wird dadurch obsolet. Die zusätzliche Trennung der Lokalspeicher in Befehls- und Datenspeicher mit je einer eigenen Busverbindung erlaubt darüberhinaus automatisch den Schutz des Befehlsspeichers vor versehentlichen Schreibzugriffen des Programms.

Optimierung der Leistungsaufnahme. Durch FPGA-basiertes Space-Sharing und Software-First-Design[21] kann die Speichergröße in jedem PSM an den Speicherbedarf der jeweiligen zugeordneten Task angepasst werden, wodurch die dynamische Verlustleistung und damit der Energieverbrauch des Rechensystems zusätzlich positiv beeinflusst werden kann[22].

4.5 Implementierung im FPGA

Inhalt. Die Flexibilität von FPGA-Hardware stellt eine wichtige Grundlage für die Anwendung von Space-Sharing dar. In diesem Abschnitt werden die Architektur des Multiprozessorsystems und mein Vorschlag zur Realisierung von Space-Sharing in FPGA-Hardware am Beispiel der prototypischen Implementierung in FPGAs der Firma Xilinx beschrieben.

4.5.1 System-Design

Hardware-First-Design. Die Umsetzung von Steuerungsaufgaben erfolgt in der klassischen Form mit Hilfe von eingebetteten Mikrocontrollern, die ein Steuerungsprogramm mit der gewünschten Funktionalität ausführen. Die Funktionalität des Systems wird einzig durch die auszuführende Software bestimmt. Als Ausgangsbasis für die Software dient jedoch immer die ausführende Rechenhardware, also der Mikrocontroller selbst. Die Hardware ist softwareunabhängig und i.d.R. auch unveränderlich, weshalb das Anwenderprogramm durch entsprechende Compiler und Code-Bibliotheken so übersetzt werden muss, dass der Mikrocontroller das Programm auch ausführen kann. Aus diesem Grund kann die klassische Vorgehensweise als *Hardware-First-Design* bezeichnet werden (vgl. **Abbildung 4.8**).

[20] indirekter Zugriff auf Speicherinhalte anderer PSM durch nachrichtenbasierte Interprozessorkommunikation erlaubt, vgl. 4.5.5 Interprozessorkommunikation, S. 70 f.

[21] vgl. 4.5.1 System-Design, S. 60 ff.

[22] vgl. 6.2.4 Einfluss der Lokalspeicher, S. 184 f.

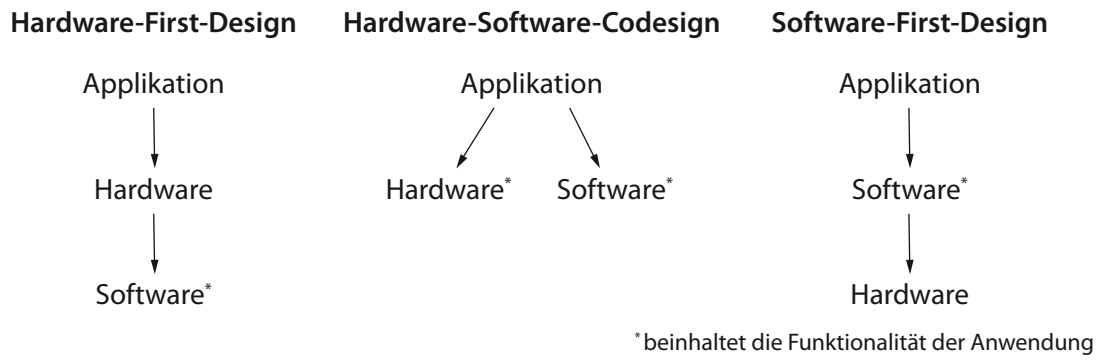


Abbildung 4.8: Verschiedene Methoden im System-Design

Hardware-Software-Codesign. In bestimmten Fällen lassen sich einzelne Funktionen besser in Hardware als in Software realisieren. Diese Möglichkeit verfolgt das *Hardware-Software-Codesign*, bei dem das System zunächst anhand der Funktionalität in Hard- und Softwarekomponenten partitioniert wird (vgl. Abbildung 4.8). Die Realisierung von Steuerungsaufgaben in Hardware oder Software weist sehr unterschiedliche Charakteristiken auf. Hardwarefunktionen sind sehr schnell in der Ausführung, eignen sich aber eher für Aufgaben mit geringer Komplexität. Rechenoperationen wie z.B. trigonometrische Funktionen oder variable Schleifen sind allein in Hardware praktisch kaum zu realisieren. Nachträgliche Änderungen in der Funktionalität der Hardware sind (abgesehen von FPGAs) ebenfalls nur schwer oder gar nicht realisierbar. Softwarefunktionen sind dagegen sehr flexibel und können beliebig oft angepasst werden. Die Komplexität der Aufgabe hat lediglich einen Einfluss auf die Ausführungszeiten, die bei der Ausführung in Software allerdings prinzipbedingt sehr viel größer sind als bei der Ausführung in Hardware. Mit dem vermehrten Einsatz und der raschen Weiterentwicklung von FPGAs verwischen die Grenzen zwischen Hard- und Software jedoch zunehmend. Zudem stehen für die Entwicklung im Hardware-Software-Codesign eigene Sprachen wie z.B. SystemC[23] zur Verfügung, die eine Beschreibung und Simulation des Systemverhaltens auf einem höheren Abstraktionsniveau als mit VHDL ermöglichen. Die Realisierung des Hardware-Software-Codesigns mit FPGA-Technologie kombiniert die Geschwindigkeit von Hardware mit der Flexibilität von Software. Auf diese Weise lassen sich im Codesign Aufgaben in einer Kombination aus Hard- und Software ausführen[24].

[23] Modellierungssprache für die Entwicklung elektronischer Systeme, die sowohl Hardware- als auch Softwarekomponenten enthalten, vgl. [Bl10]

[24] vgl. [Ge07, S. 203 ff.]

Software-First-Design. Die funktionale Realisierung von Steuerungsaufgaben erfolgte in der Vergangenheit entweder durch eine reine Software-Lösung für bereits existierende Hardware oder durch eine Hardware/Software-Partitionierung der Aufgabe im Rahmen eines Hardware-Software-Codesigns. Für die Realisierung von Space-Sharing wird nun eine dritte Methodik eingeführt, bei der das Steuerungsprogramm (bzw. die Funktionssoftware im automobilen Steuergerät) als Ausgangsbasis für den Entwurf der ausführenden Rechenhardware dient. Diese neue Methodik des System-Designs wird von mir nachfolgend als *Software-First-Design* bezeichnet (vgl. Abbildung 4.8).

Beim Software-First-Design wird die Funktionalität (z.B. des Steuergerätes) mit Hilfe einer Programmiersprache (z.B. mit C) in einer Softwarelösung realisiert. Bei einem Hardware-First-Design würde nun die Funktionssoftware für eine geeignete aber bereits vordefinierte und unveränderbare Hardware kompiliert werden. Im Software-First-Design dagegen wird nun auf Basis der Funktionssoftware eine geeignete Rechenhardware generiert und im FPGA implementiert. Auf diese Weise kann die Rechenhardware so an die Bedürfnisse der Software angepasst werden, dass eine optimale Verarbeitung der Daten erfolgt. Zum Beispiel kann die Prozessorarchitektur an den verwendeten Befehlen ausgerichtet werden. Das heißt, wenn z.B. im Programm keine Fließkommaoperationen ausgeführt werden, dann wird keine Fließkommaeinheit implementiert, was neben dem Bedarf an Chip-Ressourcen[25] auch den Energieverbrauch[26] erheblich reduziert.

Bei einem Hardware-Software-Codesign wird ebenfalls neben Software auch Hardware generiert. Der Unterschied zum Software-First-Design besteht darin, dass die Funktionalität der Anwendung gleichberechtigt sowohl in Hardware als auch Software realisiert werden kann, je nachdem wo sich die Funktion besser umsetzen lässt. Im Software-First-Design dient die generierte Hardware lediglich der Ausführung der Funktionssoftware. Das heißt, die Funktionalität der Anwendung wird hier (wie auch im Hardware-First-Design) ausschließlich durch Software realisiert. Eine Ausnahme bildet lediglich der Einsatz von Koprozessoren, mit denen sehr rechenintensive Funktionen wie z.B. die FFT in Hardware implementiert werden können, um die Rechenleistung des Prozessors anwendungsspezifisch zu verbessern.

[25] vgl. Abbildung 4.13: Anteiliger Flächenbedarf im Softprozessor, S. 67

[26] vgl. 6.2.6 Einfluss des Prozessordesigns, S. 186 f.

Partitionierung der Software. Das Prinzip von Space-Sharing macht es erforderlich, die Applikationssoftware in nebenläufige Teile zu partitionieren, damit beim anschließenden Entwurf der Rechenhardware jeder Partition ein eigenes Prozessor-Speicher-Modul[27] (PSM) zugewiesen werden kann. Die **Abbildung 4.9** zeigt eine solche Partitionierung am Beispiel eines bereits bestehenden Anwenderprogramms mit drei nebenläufig ausgeführten Tasks, welches auf einem konventionellem Rechensystem bereits existiert. Oben links ist als Ausgangspunkt die Speicherverteilung in einem gemeinsamen Speicher dargestellt, wie sie in einem System mit Time-Sharing realisiert wäre. Daneben wird auf der rechten Seite das Anwenderprogramm in drei nebenläuige Teile partitioniert, wobei die Anzahl und die Größe der Partitionen von der jeweiligen Task abhängen. Abschließend wird im letzten Schritt für jede Partition ein eigenes PSM im FPGA erzeugt, welches aus einem Softprozessor plus Lokalspeicher besteht.

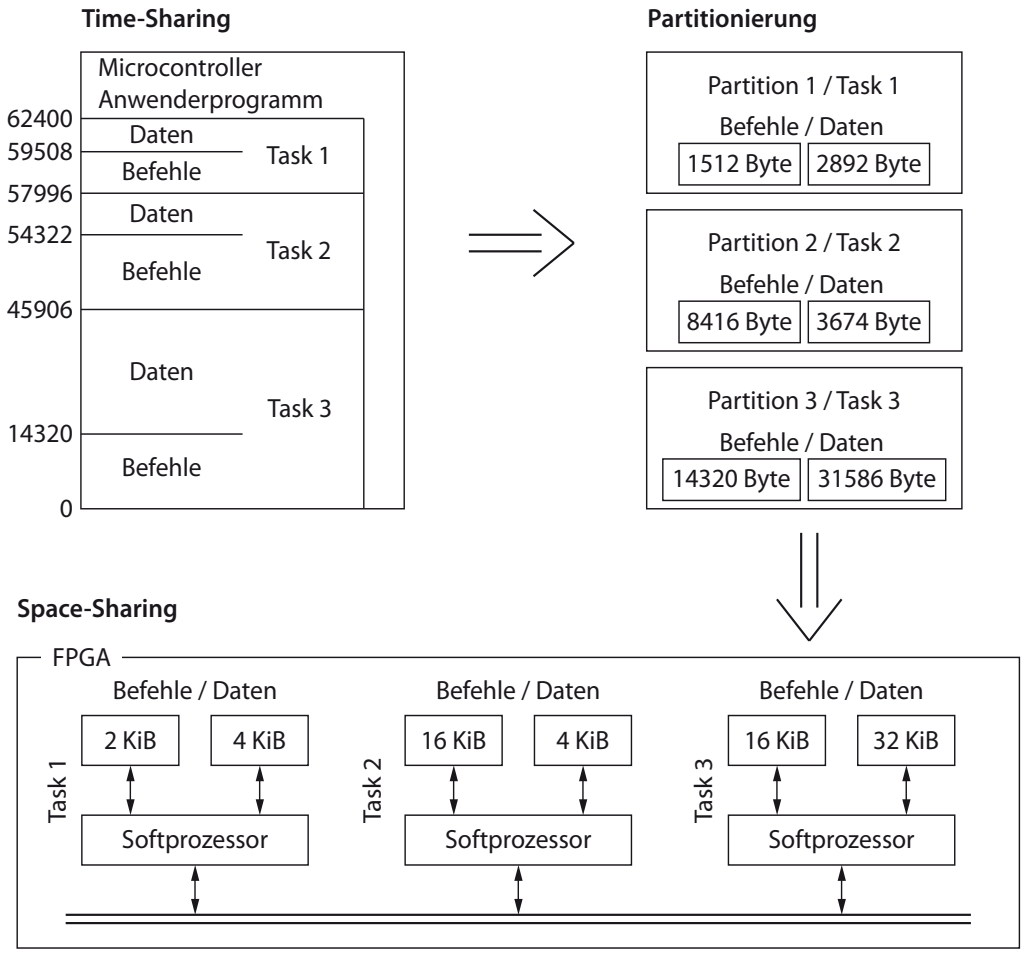


Abbildung 4.9: Partitionierung bestehender Anwenderprogramme

[27] vgl. Abbildung 4.3: Lose gekoppeltes System aus Recheneinheiten (PSM), S. 52

Systementwurf. Die Vorgehensweise beim Systementwurf nach dem *Software-First-Design* ist in der **Abbildung 4.10** dargestellt. Den Ausgangspunkt bildet das Anwenderprogramm, welches die Funktionalität der Applikation beinhaltet. Zunächst erfolgt eine Teilung des Anwenderprogramms in nebenläufig ausführbare Partitionen. Aus dieser Partitionierung werden verschiedene Parameter für die Generierung der programmausführenden Rechnerarchitektur abgeleitet. Solche Parameter sind zum Beispiel die Anzahl der Prozessoren, welche sich aus der Anzahl der Partitionen ableitet, oder die Größen der Lokalspeicher, welche von der Größe der Partitionen abhängen. Auf der Basis dieser Parameter und der Spezifikation des umgebenden Systems wird eine geeignete Rechnerarchitektur in VHDL-Code erzeugt. Durch die anschließende Synthese wird diese Rechnerarchitektur in FPGA-Hardware übersetzt und zusammen mit dem kompilierten Programm als Bitstream in den FPGA geladen und ausgeführt.

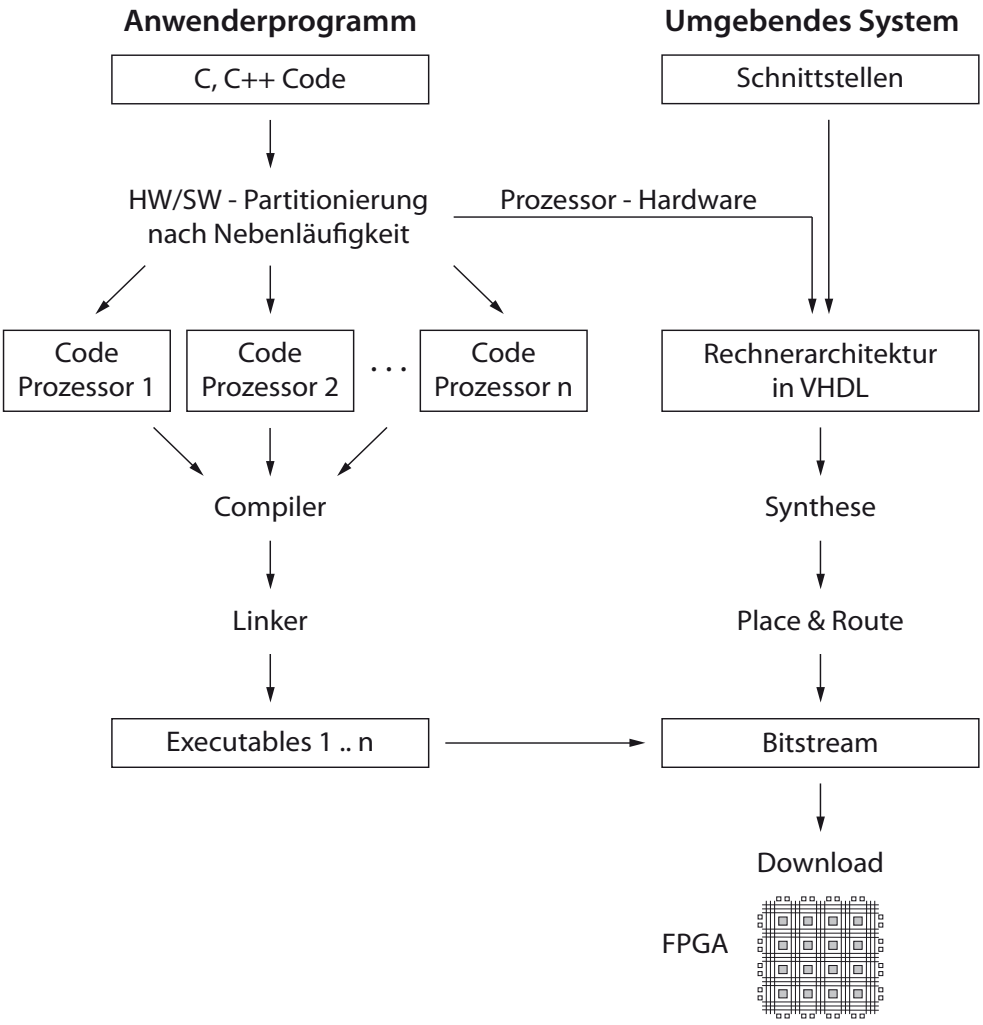


Abbildung 4.10: Systementwurf mit Space-Sharing und Software-First-Design

Entwicklungsumgebung. Für die prototypische Implementierung von *ConPar* wurde das *Embedded Development Kit (EDK)* der *ISE Design Suite* der Fa. Xilinx in der Version 10.1 verwendet. Die Entwicklungsumgebung EDK bietet in der Version 10 die Möglichkeit, Multiprozessorsysteme in der Hardwarebeschreibungssprache VHDL zu entwickeln und unter Verwendung der Programmiersprachen C und C++ zu programmieren [vgl. Xi02]. In späteren Versionen der ISE Design Suite erfolgt mit dem EDK nur noch die Synthese der Rechenhardware. Die Programmierung der Soft-Prozessoren erfolgt ausschließlich mit dem *Software Development Kit (SDK)* der Fa. Xilinx. Auf das Software-First-Design hat diese strikte Trennung von Hardware- und Software-Entwicklung jedoch keine Auswirkung.

4.5.2 Realisierbarkeit komplexer MPSoCs mit FPGAs

Technische Historie von FPGAs. Die maximale Größe des MPSoCs wird durch die Anzahl der verfügbaren Logikzellen und durch die Größe des verfügbaren Speichers auf dem FPGA bestimmt. In den letzten zehn Jahren hat die FPGA-Technologie bei beiden Parametern, wie auch bei der Anzahl der externen I/Os, der maximalen Taktfrequenz und dem Energieverbrauch, eine enorme Entwicklung erfahren, die über das Moore'sche Gesetz[28] hinausgeht. Die **Abbildung 4.11** zeigt die technische Entwicklung von FPGAs am Beispiel der verfügbaren Logikzellen in Xilinx FPGAs aus den Jahren 2003 bis 2011. Einen fast identischen Verlauf zeigt die Entwicklung der verfügbaren Speichergröße, welche in der **Abbildung 4.12** dargestellt ist. Eigene Untersuchungen an den dargestellten FPGAs vom Typ Spartan-3 XC3S1000[29] und Virtex-4 XC4VFX100[30] haben gezeigt, dass maximal 6 (bei Spartan-3) bzw. 34 (bei Virtex-4) Standard-Softprozessoren vom Typ MicroBlaze [Xi08b] plus Lokalspeicher implementiert werden können. In [Mp08] ist ein MPSoC mit 80 solcher Softprozessoren auf einem Virtex-5 FPGA beschrieben. Rechnet man diese Zahlen weiter hoch, so könnte ein Virtex-7 FPGA der aktuellen Generation[31] mehr als 700 Softprozessoren aufnehmen, wobei die tatsächliche Zahl aufgrund der ebenfalls gestiegenen Effektivität durch verbesserte LUTs sogar noch größer ausfallen dürfte.

[28] Gordon Moore (geb. 1929), Mitbegründer der Fa. Intel und Autor des Moore'schen Gesetzes, wonach sich die Integrationsdichte alle 24 Monate verdoppelt

[29] auf einem Spartan-3 Starter Kit Board der Fa. Digilent [Xi05]

[30] auf einem XpressFX Board der Fa. PLDA [Pl06]

[31] Xilinx Produktinformation, Stand: 2011 [Xi11b, S. 1]

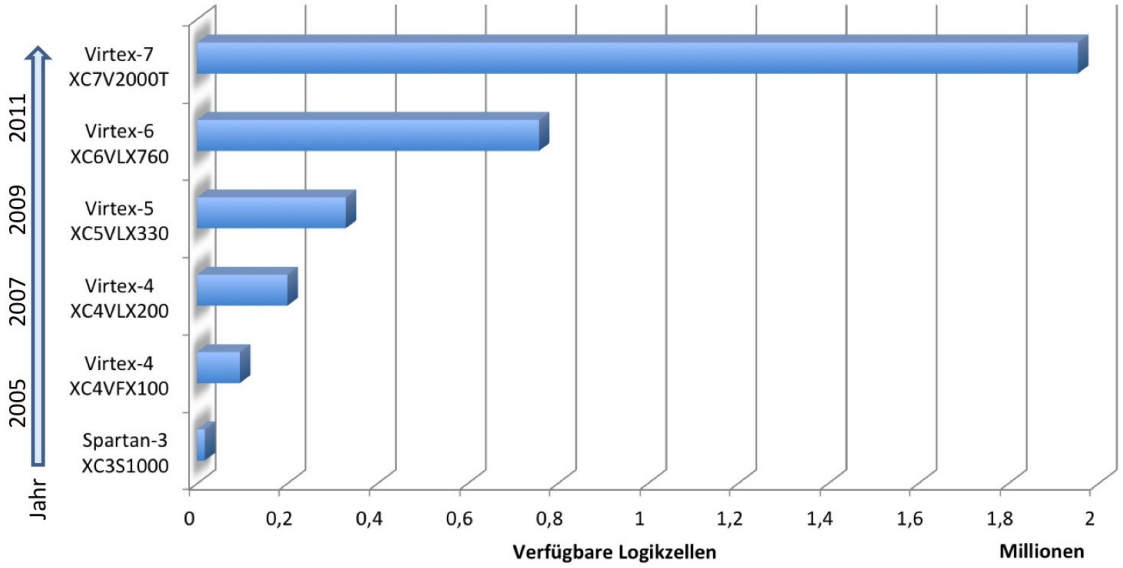


Abbildung 4.11: Technische Entwicklung bei Xilinx FPGAs anhand der Logikzellen

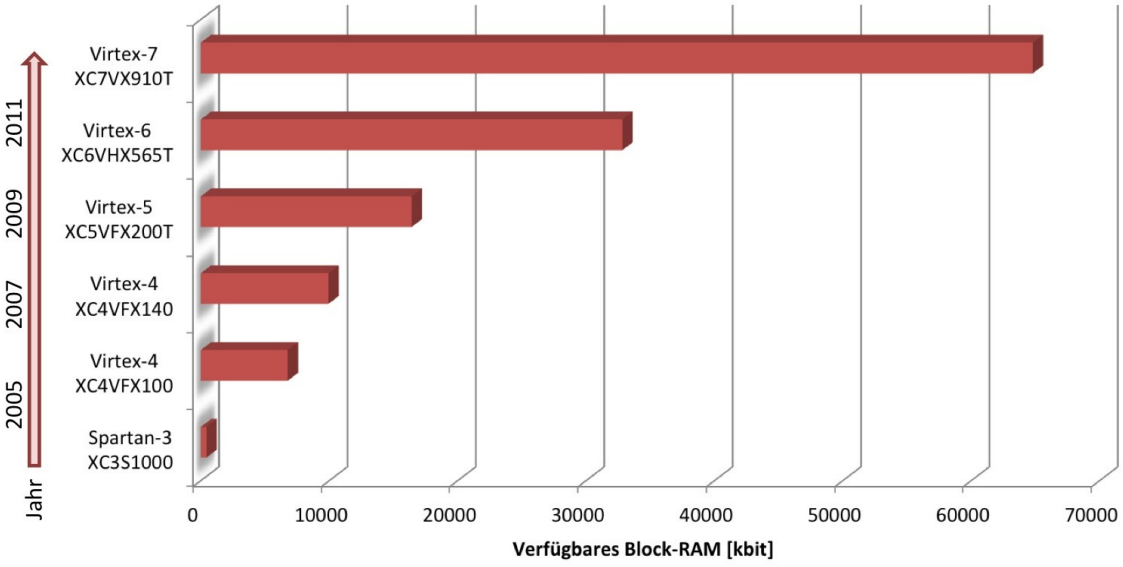


Abbildung 4.12: Technische Entwicklung bei Xilinx FPGAs anhand des Speichers

Zukünftige technische Entwicklung von FPGAs. Die äußerst rasante Entwicklung der FPGA-Technologie in der Vergangenheit lässt viel Raum für Spekulationen über zukünftige Trends. Neue FPGA-Technologien wie *Stacked Silicon*[32] oder die weitere Reduzierung der Strukturgröße[33] lassen auch in der Zukunft deutliche Fortschritte bei wichtigen Parametern wie Logik, Speicher, Taktrate und Energieeffizienz von FPGAs erwarten.

[32] Xilinx Stacked Silicon Interconnect Technology [Do10]

[33] derzeit 28 nm, vgl. Altera [Al10] und Xilinx [Wu11]

4.5.3 Softprozessoren

Xilinx MicroBlaze. Für die prototypische Implementierung der PSM werden proprietäre Softprozessoren der Fa. Xilinx vom Typ MicroBlaze[34] verwendet. MicroBlazes sind 32-Bit RISC-Prozessoren, deren Architektur als IP-Core zur Verfügung steht. Der große Vorteil von FPGA-basierten Softprozessoren besteht darin, dass der Anwender den Umfang der Prozessorarchitektur weitestgehend selbst bestimmen kann. So kann im EDK mit einigen wenigen Einstellungen per Mausklick festgelegt werden, welche Befehlspipeline, Rechenwerke, Caches oder Prozessorbusanbindungen implementiert werden sollen.

Schlanke Prozessorarchitektur. Das Prinzip von Space-Sharing basiert darauf, Software in Nebenläufigkeit auf einer möglichst großen Anzahl von Prozessoren auszuführen, die auf einem FPGA implementiert sind. Da die zur Verfügung stehende Chipfläche des FPGAs begrenzt ist, sollte jeder Prozessor möglichst wenig Fläche für sich beanspruchen. Das bedeutet, dass die Prozessorarchitektur auf die notwendigen Funktionalitäten beschränkt werden sollte. Die **Abbildung 4.13** zeigt den von mir ermittelten anteiligen Ressourcenverbrauch am Beispiel eines Softprozessors vom Typ Xilinx MicroBlaze. Deutlich zu sehen ist hier z.B. der mit 38 % sehr hohe Ressourcenverbrauch der Fließkommaeinheit (*Floating Point Unit*). Die **Abbildung 4.14** zeigt darüberhinaus den unterschiedlichen Ressourcenverbrauch eines Xilinx MicroBlaze Softprozessors in der minimalen und in der maximalen Ausbaustufe. Auch hier sind die Unterschiede sehr deutlich zu erkennen.

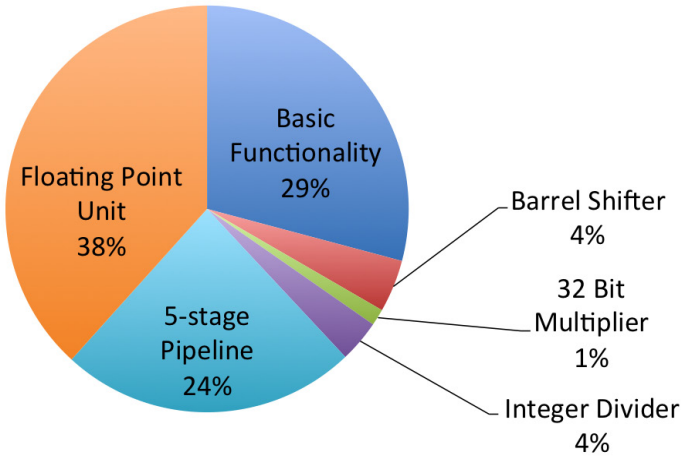


Abbildung 4.13: Anteiliger Flächenbedarf im Softprozessor[35]

[34] vgl. [Xi08b]

[35] exemplarisch ermittelt an einem Xilinx MicroBlaze mit Spartan-3 FPGA

Deterministische Programmausführung. Der Verzicht auf leistungssteigernde aber technisch aufwändige Elemente wie z.B. Caches oder Out-of-Order-Execution[36] trägt darüber hinaus zur deterministischen Programmausführung und damit zur Echtzeitfähigkeit des Parallelrechners bei. Davon unabhängig können zur Steigerung der Rechenleistung des Softprozessors applikationsabhängig zusätzliche Recheneinheiten (z.B. Fließkommaeinheit) oder Koprozessoren in Form von IP-Cores eingebunden werden, die zwar zusätzliche Chipfläche benötigen, aber nicht die Echtzeitfähigkeit gefährden.

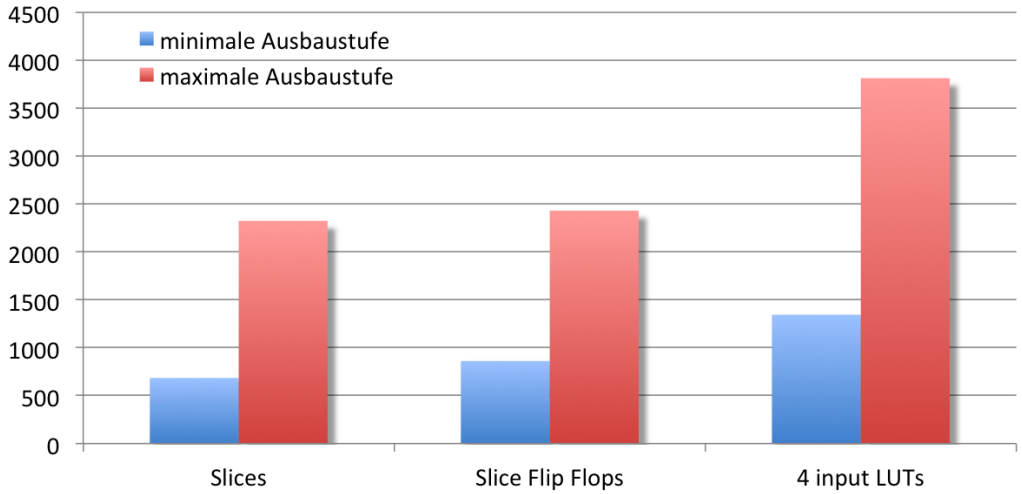


Abbildung 4.14: Minimaler und maximaler Ressourcenverbrauch im Softprozessor[37]

Bedingte Sprünge. Die Architektur der Befehlspipeline des RISC-Prozessors führt bei bedingten Sprüngen im Programm auch ohne Sprungvorhersage zu einem unbestimmbaren Verhalten, da bereits vor der Sprungsausführung neue Befehle in die Pipeline geladen werden. Ohne Sprungvorhersage wird immer davon ausgegangen, dass der Sprung nicht genommen wird. Ein genommener Sprung führt deshalb zu einer Invalidierung der Befehlspipeline, was zu nicht vorhersagbaren Verzögerungen (bzw. Jitter) bei der Programmausführung führt, da die Pipeline erst wieder neu gefüllt werden muss. Es gilt, je länger die Pipeline, desto größer die Verzögerung. Xilinx löst dieses Problem beim MicroBlaze durch die Einführung von *Delay Slots* in der Pipeline, welche die Verzögerungen aufgrund von bedingten Sprüngen in Verbindung mit einer sehr kurzen 3-stufigen Befehlspipeline auf nur einen einzigen Prozessortakt reduzieren[38].

[36] vgl. 3.1.4 Echtzeitfähige Hardware, S. 31 f.

[37] exemplarisch ermittelt an einem Xilinx MicroBlaze mit Spartan-3 FPGA

[38] vgl. [Xi08b, S. 44-45]

4.5.4 Speicher

FPGA-interner Speicher. Wie die Abbildung 4.12, S. 66 zeigt, verfügen moderne FPGAs über bis zu 64.800 kbit an internem Speicher, was für ein eingebettetes SoC bereits enorm viel Speicherplatz darstellt. Der interne Speicher wird als Lokalspeicher direkt an den Prozessor gekoppelt, wobei die Speichergröße abhängig vom FPGA-Typ in bestimmten Schrittweiten wählbar ist und somit an die jeweilige Softwarepartition angepasst werden kann. Typische Speichergrößen befinden sich im Bereich von 8 bis 512 kbit, wobei im Xilinx EDK max. 4 Gbit einstellbar sind.

Harvard-Architektur. Bei Space-Sharing im FPGA verfügt jedes PSM über einen eigenen Speicher, der als internes Block-RAM realisiert ist. Xilinx erlaubt darüber hinaus, diesen internen Speicher als Harvard-Architektur zu realisieren[39]. Wie in der **Abbildung 4.15** zu sehen ist, besitzen die Lokalspeicher der Prozessoren auch getrennte Speicherbusverbindungen[40] für Daten und Befehle. Durch die räumliche Isolation wird nicht nur der Speicherschutz zwischen den Lokalspeichern der Prozessoren, sondern auch zwischen den Befehls- und Datenspeichern eines Prozessors hergestellt. Zudem existiert auch keine zeitliche Beeinflussung der unterschiedlichen Speicherzugriffe zueinander.

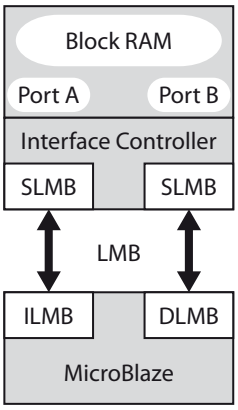


Abbildung 4.15: Anbindung des Lokalspeichers an MicroBlaze

Externer Speicher. Neben dem internen Speicher besteht die Möglichkeit, externe Speichermodule an den Prozessor zu koppeln. Dies geschieht jedoch über den allgemeinen Prozessorbus, benötigt relativ viele I/O-Verbindungen zur Peripherie und muss in realer zusätzlicher Speicherhardware ausgeführt werden, so dass sich Space-Sharing auf den internen Speicher im FPGA beschränkt.

[39] vgl. [Xi08b, S. 46]

[40] Local Memory Bus, kurz LMB [vgl. Xi08b, S. 77-82]

4.5.5 Interprozessorkommunikation

Nachrichtenaustausch. Die Interprozessorkommunikation ist eines der entscheidenden Merkmale, damit das Prinzip von Space-Sharing in einem Steuergerät funktioniert. Der Hintergrund besteht darin, dass rezentralisierte und nebenläufige Softwarekomponenten zwar zeitlich unabhängig voneinander ausgeführt werden können, trotzdem aber eine Interaktion zwischen Softwarekomponenten in Form von Nachrichtenaustausch stattfinden muss, damit das System funktioniert.

Echtzeitanforderung. Aufgrund der Echtzeitanforderungen an das Rechensystem muss auch die systemweite Kommunikation auf dem FPGA in Echtzeit erfolgen. Ein gemeinsamer Prozessorbus scheidet an dieser Stelle aufgrund mangelnder Skalierbarkeit aus. Alternativ bietet sich die Möglichkeit, einen direkten Kommunikationskanal zwischen Sender und Empfänger der Nachricht zu verwenden, um ein Netzwerk zur Interprozessorkommunikation aufzubauen. Hierfür eignet sich z.B. das FSL-Interface[41] des Xilinx MicroBlaze, welches eine schnelle unidirektionale Punkt-zu-Punkt-Verbindung zwischen zwei Prozessoren zur Verfügung stellt (**Abbildung 4.16**) und bei vorhergehenden Arbeiten[42] bereits erfolgreich im MPSoC eingesetzt wurde.

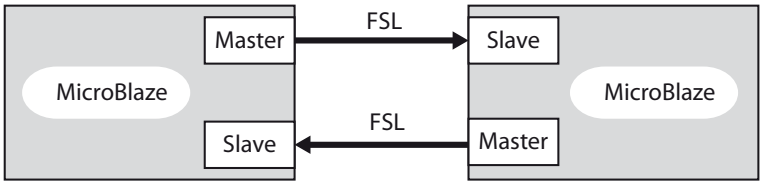


Abbildung 4.16: Bidirektionale Kommunikation mit zwei FSL-Verbindungen

4.5.6 Externe Kommunikation

Getrennte und gemeinsame Ressourcen. Bei der externen Kommunikation gibt es zwei Fälle, die gesondert betrachtet werden müssen. Im einfachen Fall existiert eine Kommunikationsverbindung nach außen, die nur von einem einzigen Prozessor bedient wird. Dieser Fall stellt keine besonderen Anforderungen an die zugrundeliegende Hardware. Der Prozessor empfängt die Daten und stellt diese anderen Prozessoren im MPSoC zur Verfügung. Schwieriger wird die Kommunikation im

[41] Fast Simplex Link, kurz FSL [vgl. Xi08b, S. 83 f.]

[42] z.B. in [Hu05] und [De06]

Hinblick auf die Echtzeitfähigkeit und die Skalierbarkeit des Systems bei gemeinsamen Zugriffen auf eine externe Verbindung. Hier muss der Zugriff nach außen über das gemeinsame Verbindungsnetzwerk erfolgen, an dem die externe Kommunikation als IP-Core angebunden wird (**Abbildung 4.17**).

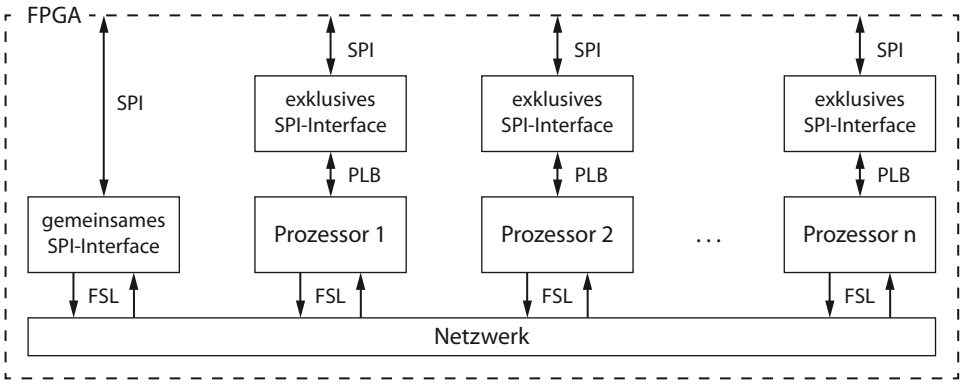


Abbildung 4.17: Beispielanbindung externer Kommunikation mit SPI

4.6 Zusammenfassung

Echtzeitaufgaben beliebiger Komplexität. Space-Sharing definiert eine Rechnerarchitektur auf der Basis eines lose gekoppelten bzw. nachrichtenbasierten Multiprozessorsystems in Kombination mit einer neuen Design-Methodik, dem *Software-First-Design*. Die Motivation für das neue (*n*,*n*)-Space-Sharing besteht darin, ein skalierbares Rechensystem in Form eines Echtzeitparallelrechners zu konstruieren, welches für Echtzeitaufgaben bestehend aus beliebig vielen nebenläufigen Teilaufgaben geeignet ist.

Partitionierung von Software. Wie bei konventionellen Parallelrechnern, so gilt auch bei Space-Sharing die Granularität der Aufgabe als notwendiges Kriterium. Das heißt, die Software muss in nebenläufig ausführbare Tasks partitioniert und so auf mehrere Prozessoren verteilt werden können. Im Falle der Rezentralisierung von Steuergeräten ergibt sich die Granularität aus der Anzahl der Teilaufgaben im Fahrzeug, welche in der bestehenden E/E-Architektur auf mehreren Steuergeräten verteilt ausgeführt werden.

Statische Zuordnung. Ein weiteres notwendiges Kriterium für Space-Sharing besteht darin, dass die Tasks den Prozessoren fest zugeordnet sind. Das bedeutet, dass zur Laufzeit keine neuen Prozesse oder Threads erzeugt werden können, was bei eingebetteten Steuerungen jedoch den Regelfall darstellt.

Räumliche und zeitliche Isolation. Neben der Skalierbarkeit besteht ein weiterer Vorteil von Space-Sharing in der Isolation der Tasks untereinander. Durch den Einsatz von Prozessor-Speicher-Modulen (PSM) erfolgt eine räumliche Isolation der Lokalspeicher, so dass der Speicher jeder Task vor unbeabsichtigten Schreibzugriffen anderer Tasks automatisch geschützt ist. Durch die statische Zuordnung jeder Task auf einen eigenen Prozessor erfolgt auch eine zeitliche Isolation, so dass die Laufzeit einer Task nicht durch andere Tasks, z.B. durch Unterbrechung aufgrund höherer Prioritäten, beeinflusst wird.

Software-First-Design auf FPGA-Basis. Das Prinzip von Space-Sharing beruht darauf, jeder Task einen eigenen Prozessor nebst Speicher zur Verfügung zu stellen. Aus diesem Grund ist die Architektur der Rechenhardware zwingend auf die auszuführende Software ausgerichtet. Die softwareabhängige Gestaltung der Rechenhardware ist Bestandteil einer neuen Design-Methodik, dem *Software-First-Design*. Die Basis für diese Methodik bildet die FPGA-Technologie, welche die Generation von Rechenhardware auf der Grundlage bestimmter Parameter aus der Software heraus erlaubt. Auf diese Weise ist es möglich, die Rechenhardware aus den Anforderungen der Applikationssoftware zu generieren.

Kapitel 5 - Interprozessorkommunikation

Inhalt. Die Interprozessorkommunikation stellt bei Parallelrechnern einen ganz wesentlichen funktionalen Aspekt dar. Es existieren bereits viele unterschiedliche Lösungen, bei denen jedoch bisher nur Kriterien wie die Skalierbarkeit und die Bandbreite, nicht aber die Echtzeitfähigkeit des Systems berücksichtigt werden. In diesem Kapitel soll deshalb die Interprozessorkommunikation als wesentlicher Bestandteil des Echtzeitparallelrechners diskutiert werden, wobei in den nachfolgenden Ausführungen verschiedene Netzwerktopologien als Mittel zur echtzeitfähigen Interprozessorkommunikation in einem MPSoC betrachtet werden.

5.1 Grundlagen

Bedeutung für Space-Sharing. Die Echtzeitfähigkeit der Interprozessorkommunikation ist bei einem Konzept wie Space-Sharing von entscheidender Bedeutung für die Echtzeitfähigkeit des gesamten Rechensystems. Um die Echtzeitfähigkeit des FPGA-basierten Multiprozessorsystems sicherzustellen, muss das Kommunikationsverhalten der Prozessoren untereinander vorhersagbar sein. Insbesondere bei der Rezentralisierung von mehreren kooperativ arbeitenden Steuergeräten muss die Einhaltung zeitlicher Grenzen garantiert werden.

Skalierbarkeit. In Abhängigkeit von der Granularität der Aufgabe wird bei Space-Sharing im Vergleich zu konventionellen Rechensystemen eine relativ hohe Anzahl an Prozessoren verwendet[1]. Aufgrund der hohen Prozessorzahl stellt die Interprozessorkommunikation zunächst einen Engpass im Sinne des von-Neumann-Flaschenhalses dar, weshalb an dieser Stelle eine skalierbare Kommunikationslösung gefunden werden muss. Ein gemeinsamer Prozessorbus ist aufgrund der mangelnden Bandbreite und schlechten Skalierbarkeit schnell überfordert. Punkt-zu-Punkt-Verbindungen können zwar eine hohe Bandbreite abdecken, sind aber aufgrund der mit $O(N^2)$ steigenden Komplexität bei einer entsprechend hohen Prozessorzahl N nicht mehr realisierbar. Erforderliche Parameter wie Skalierbarkeit und Bandbreite können also weder durch Punkt-zu-Punkt-Verbindungen noch durch einen gemeinsamen Prozessorbus abgedeckt werden, weshalb in den nachfolgenden Abschnitten nach geeigneten Verbindungsstrukturen gesucht wird.

[1] vgl. 4.1.3 Zuordnung von Prozessoren, S. 48 ff.

Implementierbarkeit. Aufgrund der bisherigen Entwicklung und der inzwischen ausreichend hohen Integrationsdichte moderner FPGAs[2] soll der Echtzeitparallelrechner *ConPar* als Multiprozessorsystem on-Chip (MPSoC) auf einem einzigen FPGA-Chip implementiert werden. Zu diesem Zweck kommen allerdings nur Netzwerkstrukturen in Frage, die als Network-on-Chip (NoC) auf demselben Chip implementiert werden können. Einige NoC-Varianten wurden bereits in [Be02] und [At08] als neues Paradigma der Kommunikation in SoCs eingeführt. Eine vergleichende Übersicht der Eigenschaften von NoCs gegenüber einfachen Busverbindungen und Punkt-zu-Punkt-Lösungen findet sich in [Le07].

Echtzeitfähigkeit. Neben der Skalierbarkeit und der Integrationsfähigkeit spielt bei *ConPar* die Echtzeitfähigkeit eine ganz entscheidende Rolle. In der Vergangenheit wurde diese Eigenschaft jedoch weder bei der Konstruktion von Parallelrechnern noch bei der Konstruktion von NoCs berücksichtigt. Die Echtzeitfähigkeit des Verbindungsnetzwerks stellt deshalb eine neue Herausforderung für den Entwurf des Echtzeitparallelrechners und der Realisierung von Space-Sharing dar.

5.2 Statische Netze

Inhalt. In diesem Abschnitt werden die wesentlichen Eigenschaften von statischen Netzen hinsichtlich ihrer Eignung für die Interprozessorkommunikation im Echtzeitparallelrechner analysiert. Dabei liegt der Schwerpunkt der Betrachtung auf den Eigenschaften der Echtzeitfähigkeit und der Skalierbarkeit.

5.2.1 Stand der Technik

Verwendung als NoC. In der Vergangenheit wurden bereits zahlreiche Arbeiten zum Thema der FPGA-basierten Multiprozessorsysteme on-Chip (MPSoC) veröffentlicht, wobei ausschließlich statische Verbindungsstrukturen verwendet wurden. Eine bewertende Analyse zu voll vermaschten, ring- und sternbasierten Topologien auf Xilinx FPGAs ist in [Hu05] enthalten. Als Ergebnis zeigen sich eine Reihe konstruktiver und funktionaler Einschränkungen. So ist etwa die voll vermaschte Verbindungsstruktur auf 9 Knoten beschränkt, da für jeden Xilinx Softprozessor typbedingt nur maximal 8 Verbindungen zur Verfügung stehen. Demgegenüber

[2] vgl. Abbildung 4.11: Technische Entwicklung bei Xilinx FPGAs anhand der Logikzellen, S. 66

benötigt eine Ringstruktur nur jeweils 2 Verbindungen pro Knoten. Allerdings steigt mit der Größe des Rings auch die Latenz bei der Nachrichtenübermittlung. Die Sterntopologie könnte beide Probleme beheben, birgt aber gleichzeitig die Gefahr von Datenstaus an den zentral gelegenen Knoten. Vor diesem Hintergrund ist keine der genannten Topologien in der Lage, eine skalierbare und gleichzeitig echtzeitfähige Kommunikationstruktur aufzubauen. Weitere Arbeiten zu FPGA-basierten MPSoCs befinden sich u.a. in [De06], [Mp08] und [Xu08], wo ebenfalls statische Verbindungsstrukturen ihre Verwendung finden.

5.2.2 Aufbau

Übertragung der Daten über Zwischenknoten. Statische Netze besitzen Verbindungsstrukturen, die aus beliebigen mathematischen Graphen abgeleitet werden können. In einem Multiprozessorsystem stellen die Knoten die Prozessoren und die Kanten die Verbindungen zwischen den Prozessoren dar [Ri97, S. 76]. Die Topologie statischer Netze besteht allgemein in der festen und direkten Verbindung benachbarter Knoten im Netz. Aus diesem Grund werden statische Netze auch als direkte Netze bezeichnet. Direkte Netze sind dadurch gekennzeichnet, dass jeder Knoten nur mit seinen unmittelbaren Nachbarknoten direkt kommunizieren kann. Die Kommunikation zwischen Sender- und Empfängerknoten, die nicht unmittelbar miteinander benachbart sind, muss über Zwischenknoten erfolgen. Durch diesen Umstand wird jeder Knoten im Netz gleichzeitig zu einem Routerknoten, der Nachrichten von seinen Nachbarknoten empfängt und gemäß einem definierten Routingalgorithmus an einen anderen Nachbarknoten weiterleitet, sofern er nicht selbst der Empfänger der Nachricht ist. Die zusätzliche Routerfunktionalität der Knoten führt zu einer höheren Komplexität bei der Gestaltung und Implementierung der einzelnen Knoten.

Verschiedene Topologien. Statische Netze besitzen sehr viele unterschiedliche Topologien. Vor allem in Ein-Chip-Systemen (SoCs) sind statische Netze jedoch i.d.R. in einfachen und übersichtlichen Strukturen wie 2D-Gitter oder Bäume ausgeführt. In der Vergangenheit sind aber auch komplexere Topologien wie mehrdimensionale Tori und Hypercubes als NoC vorgestellt worden [Ku02, Bj06, Ze03, Be06]. Allen Topologien gemein sind die feste Verdrahtung und die Übertragung der Daten über Zwischenknoten. Die **Abbildung 5.1** zeigt beispielhaft einige der genannten Topologien.

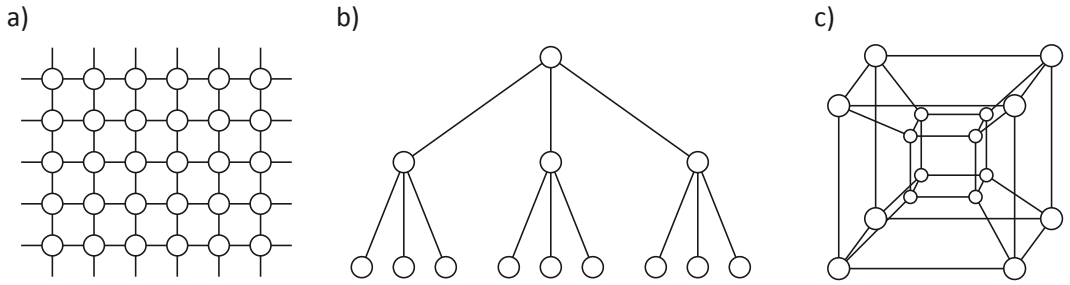


Abbildung 5.1: Topologien statischer Netze: a) 2D-Gitter, b) Baum, c) 4D-Hypercube

5.2.3 Besondere Eigenschaften

Multi-Hops. Der bei statischen Netzen erforderliche Weg der Nachricht über Zwischenknoten besitzt noch einige weitere wichtige Aspekte. Aufgrund der fehlenden direkten Verbindung zwischen Sender- und Empfängerknoten müssen statische Netze als paketvermittelnde Netze betrieben werden. Die zu übertragende Nachricht wird in kleine Pakete zerlegt (*marshalling*) und dann paketweise über Zwischen- bzw. Routerknoten vom Sender zum Empfänger durch das Netz transportiert. Dabei „hopst" das Paket von einem Knoten zum nächsten Knoten, bis der Empfängerknoten erreicht ist. Die Latenz der Nachricht ist abhängig vom Abstand zwischen Sender und Empfänger im Netz. Je größer der Abstand, desto mehr Routerknoten werden in die Übertragung eingebunden und desto größer ist die Latenz. Die Übertragung einer Nachricht über mehrere Routerknoten bezeichnet man als *Multi-Hop*. Die **Abbildung 5.2a** zeigt einige beispielhafte Pfadlängen in einem Gitternetz, deren unterschiedliche Anzahl der beteiligten Routerknoten (R) auf dem varianten Abstand zwischen Sender (S) und Empfänger (E) beruhen. Mit dem Abstand variiert auch die Latenz der Nachricht im Netz.

Hotspots. In statischen Netztopologien gibt es i.d.R. einige ausgezeichnete Routerknoten, die aufgrund ihrer besonderen Lage oder aufgrund einer besonderen Nachrichtenverteilung im Netz sehr viel mehr Nachrichten weiterleiten müssen als andere Routerknoten. Diese ausgezeichneten Knoten bezeichnet man als *Hotspots*. Die **Abbildung 5.2b** zeigt die Ausbildung eines Hotspots (rot markiert) durch das zeitgleiche Zusammentreffen mehrerer Übertragungen. An diesem Punkt kann die Nachrichtendichte im Netz so groß werden, dass der Knoten überlastet wird. Solche Situationen können dauerhaft oder auch sporadisch auftreten. In jedem Fall führen Hotspots zu einer nicht vorhersagbaren Latenz in der Nachrichtenübertragung im Netz.

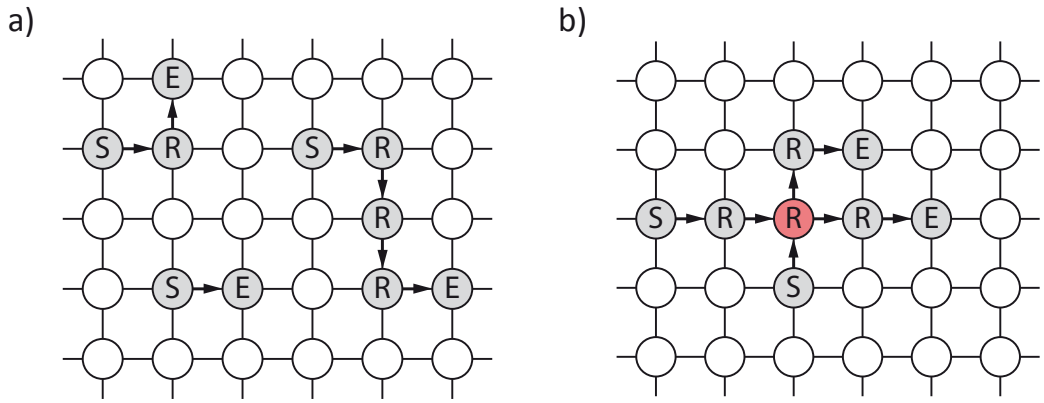


Abbildung 5.2: a) Multihop und b) Hotspot im Gitternetz

Wegewahl. Neben den topologisch bedingten Latenzen spielt die Routingstrategie eine wichtige Rolle bei der Nachrichtenübertragung, da die Anzahl der beteiligten Routerknoten neben dem Abstand auch von der gewählten Route durch das Netz abhängt. Mit dem Einsatz einer adaptiven Routingstrategie können Hotspots durch eine dynamische Wegewahl im Netz umgangen werden, was auf der anderen Seite aber meist zu einem längeren Weg durch das Netz führt.

5.2.4 Echtzeitverhalten

Knotenabstand und Latenz. Für die Betrachtung möglicher Echtzeitfähigkeiten und der Implementierung als NoC sollen die Beispiele aus der Abbildung 5.1 dienen. Links ist ein zweidimensionales Gitter dargestellt, in der Mitte ein Baum und rechts befindet sich ein vierdimensionaler Hypercube. Das Gitter besitzt den Vorteil einer sehr einfachen Konstruktion, wodurch das Routing ebenfalls recht einfach ausfällt. Der große Nachteil des Gitters besteht in der relativ hohen Anzahl an Multi-Hops. Der Abstand zwischen Sender- und Empfängerknoten im Netz variiert zwischen eins und dem Durchmesser des Netzes als Maximum (vgl. Abbildung 5.2). Die Schwankungsbreite des Abstandes wächst proportional mit der Größe des Gitters, weshalb die Skalierbarkeit im Hinblick auf die Latenz der Nachricht stark eingeschränkt ist. Zudem hängt die Latenz von der Position der Sender- und Empfängerknoten im Netz ab, da die Knoten in den Randbereichen topologisch bedingt höhere Latenzen aufweisen als die Knoten im Zentrum des Gitters, deren maximaler Abstand nur einen halben Netzdurchmesser beträgt. Erweitert man das Gitter mit Wrap-Around-Enden, so entsteht ein zweidimensionaler Torus, bei dem das Netz aus Sicht jedes Knotens unabhängig von dessen

Position immer gleich aussieht. Die Schwankungsbreite des Abstandes zwischen Sender- und Empfängerknoten ist im Vergleich zum Gitter halbiert, kann aber auch hier nicht vollständig eleminiert werden.

Ausprägung von Hotspots. Im Vergleich zum Gitter ist der maximale Durchmesser des in der Abbildung 5.1b dargestellten Baumes aufgrund der hierarchisch aufgebauten Topologie noch einmal deutlich verringert. Das bedeutet, dass bei einer Baumstruktur grundsätzlich weniger Zwischenknoten an der Nachrichtenübertragung beteiligt sind als bei einem Gitter oder Torus. Allerdings existiert auch hier eine gewisse Schwankungsbreite je nach Position von Sender- und Empfängerknoten (vgl. **Abbildung 5.3a**). Wie die **Abbildung 5.3b** zeigt, stellen die wurzelnahen Knoten des Baumes aufgrund ihrer zentralen Lage im Netz potentielle Hotspots dar, da hier die Kommunikationswege der verzweigenden Blätter und Äste zusammenlaufen. Dieser Nachteil kann prinzipiell durch eine Fat-Tree-Konstruktion kompensiert werden, in dem die Bandbreiten der wurzelnahen Verbindungen entsprechend erhöht werden. Für ein NoC ist dieses Verfahren allerdings nicht wirklich praktikabel.

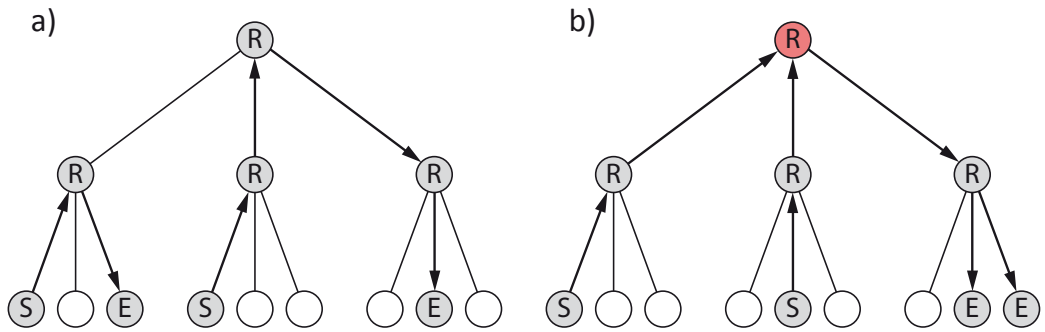


Abbildung 5.3: a) Multihop und b) Hotspot im Baum

Mehrdimensionale Netze. Im Beispiel der Abbildung 5.1c, dem 4D-Hypercube, erfolgt die Nachrichtenübertragung aufgrund des relativ geringen Durchmessers ebenfalls über sehr wenige Zwischenknoten. Das Netz sieht, wie bei einem Torus, aus Sicht der Knoten an jeder Position im Netz gleich aus. Das heißt, es existiert kein als potentieller Hotspot ausgezeichneter Knoten. Allerdings wächst bei einem Hypercube die Zahl der Dimensionen mit der Anzahl der Teilnehmer. Die kürzeren Verbindungswege werden also durch mehr Netzwerkzugänge realisiert, wobei die Anzahl der benötigten Netzwerkzugänge pro Knoten proportional mit der Dimension bzw. $O(\log_2 N)$ mit der Anzahl der Teilnehmer steigt. So sind bei dem dargestellten Hyercube mit nur 16 Knoten bereits $\log_2 N = 4$ Netzwerkzugänge

pro Knoten notwendig. Das bedeutet, dass die Verwendbarkeit von Hypercubes als NoC aufgrund der hohen Zahl an erforderlichen Netzwerkzugängen pro Knoten deutlich eingeschränkt ist.

5.2.5 Fazit

Nichtdeterministisches Verhalten. Statische Netze sind konstruktiv sehr gut skalierbar. Die mit der Knotenzahl zunehmenden Probleme aufgrund von Multi-Hops und Hotspots im Netz führen jedoch zu einer hohen Schwankungsbreite in der Latenz der Nachrichtenübertragung und damit zu einem zeitlich ungünstigen Kommunikationsverhalten. Die für *ConPar* zwingend erforderliche Interprozessorkommunikation kann deshalb nicht mit statischen Netzen realisiert werden.

5.3 Dynamische Netze

Inhalt. Dieser Abschnitt vermittelt einige Grundlagen zu mehrstufigen dynamischen Netzen, welche für das Verständnis der nachfolgenden Betrachtungen der Echtzeitfähigkeit bei der Interprozessorkommunikation nötig sind. Dabei werden hauptsächlich der Aufbau und die Funktion mehrstufiger Netze betrachtet. Abschließend erfolgt eine Einteilung mehrstufiger Netze mit Blick auf deren Echtzeitverhalten in zwei Klassen mit speziellen Eigenschaften, welche in den nachfolgenden Abschnitten 5.4 und 5.5 im Detail untersucht werden.

5.3.1 Aufbau

Schaltelement. Das wesentliche Element, welches dynamische Netze von statischen Netzen unterscheidet, ist das Schalterelement [Du03]. Während bei statischen Netzen alle Knotenverbindungen im Netz fest vordefiniert sind, können bei dynamischen Netzen Knotenverbindungen aufgrund der Schalterfunktionalität je nach Bedarf auf- und wieder abgebaut werden.

Kreuzschienenverteiler. Das funktional einfachste dynamische Netz, welches jeden Eingang mit jedem freien Ausgang zu jedem Zeitpunkt verbinden kann, ist der Kreuzschienenverteiler (*crossbar*). Allerdings steigen die Kosten mit $O(N^2)$ stark an, so dass für ein relativ kleines Netz mit $N = 8$ Ein- und Ausgängen bereits 64 Schalterelemente benötigt werden (**Abbildung 5.4**). Aus diesem Grund ist der Kreuzschienenverteiler nicht ausreichend skalierbar.

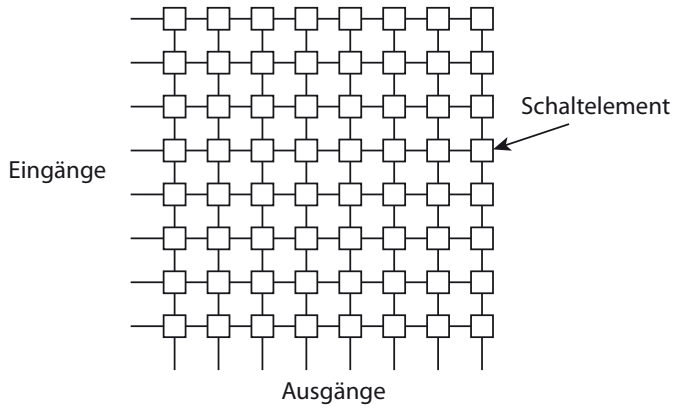


Abbildung 5.4: Kreuzschienenverteiler der Größe 8x8

Mehrstufige Netze. Eine effektive Alternative zum Kreuzschienenverteiler stellen die mehrstufigen Netze dar. Mehrstufige Netze bestehen ebenfalls aus Schaltelementen, den Kreuzschaltern der Größe 2x2. Die Kreuzschalter können grundsätzlich die beiden Stellungen *gerade* und *gekreuzt* einnehmen, wie sie in der **Abbildung 5.5** dargestellt sind. Weitere Möglichkeiten bieten der *obere Broadcast* und der *untere Broadcast* (**Abbildung 5.6**). Die Broadcastfunktionalität bietet die interessante Möglichkeit, Daten von einem Sender zeitgleich an mehrere Empfänger zu verteilen. In bidirektionalen Netzen gibt es zudem noch die Möglichkeit des *Turnaround* (Abbildung 5.6).

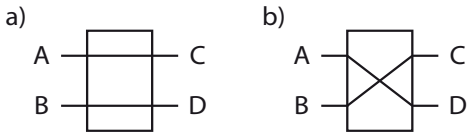


Abbildung 5.5: 2x2-Kreuzschalter: a) gerade, b) gekreuzt

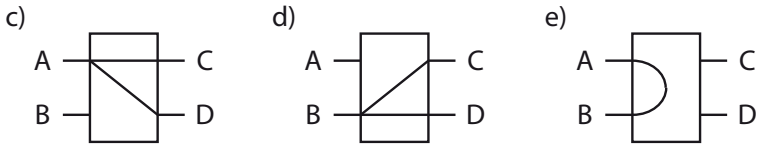


Abbildung 5.6: 2x2-Kreuzschalter: c) oberer Broadcast, d) unterer Broadcast, e) Turnaround

Schalter- und Verdrahtungsstufen. Bei mehrstufigen Netzen werden Kreuzschalter in Schalterstufen organisiert. Die Verbindung zwischen den Schalterstufen erfolgt durch Verdrahtungsstufen. Die Verdrahtungsstufen bilden Permutationen der Ausgänge einer Schalterstufe auf die Eingänge der nachgeschalteten Schalterstufe. Je nach Netztopologie variieren die Permutationen der Verdrahtungsstufen. Die **Abbildung 5.7** zeigt drei der wichtigsten Permutationen mehrstufiger Netze.

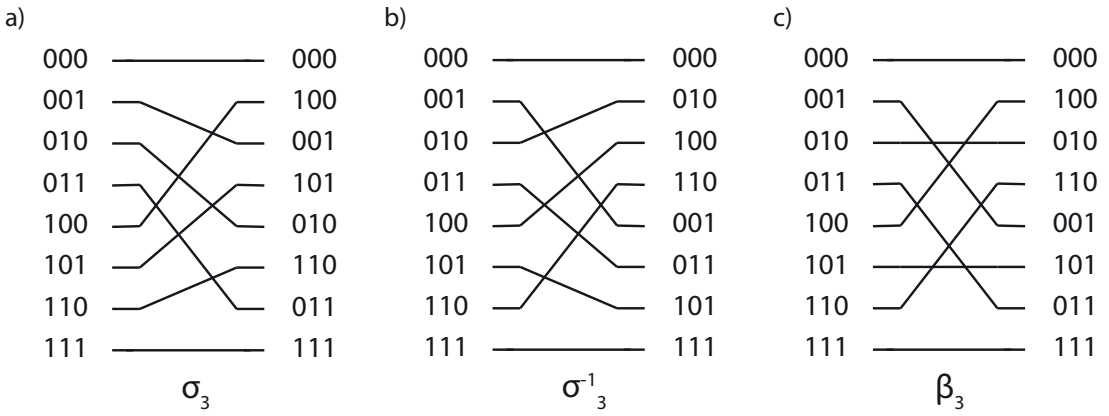


Abbildung 5.7: Permutationen für n = 3: a) Shuffle, b) Unshuffle, c) Butterfly

Vertauschung von Adressbits. Im Wesentlichen geht es in den Verdrahtungsstufen darum, die Eingänge $I = [i_n...i_2i_1]_2$ sortiert auf die Ausgänge $O = [o_n...o_2o_1]_2$ abzubilden, wodurch eine Vertauschung der Adressen stattfindet. Bei der Shuffle-Permutation (Abbildung 5.7a) findet eine Rotation der Adressbits nach links statt. Die Unshuffle-Permutation (Abbildung 5.7b) stellt die inverse (gespiegelte) Variante der Shuffle-Permutation und damit eine Rotation der Adressbits nach rechts dar. Im Falle der Butterfly-Permution (Abbildung 5.7c) wird das höchstwertige Bit (MSB) mit dem niedrigstwertigen Bit (LSB) getauscht. Mathematisch lassen sich die in der Abbildung 5.7 dargestellten Permutationen wie folgt beschreiben[3]:

$$\sigma_n(I) = [i_2 i_3 \ldots i_n i_1]_2 \qquad (5.1)$$

$$\sigma_n^{-1}(I) = [i_n i_1 i_2 \ldots i_{n-2} i_{n-1}]_2 \qquad (5.2)$$

$$\beta_n(I) = [i_n i_2 i_3 \ldots i_{n-1} i_1]_2 \qquad (5.3)$$

Beispiele für mehrstufige Netze. Die Topologie mehrstufiger Netze findet sich u.a. bei verschiedenen $\log_2$N-Netzen, deren Aufbau im Abschnitt 5.4.1 auf Seite 88 beschrieben ist. Typische Beispiele sind das Omega-Netz, das Indirect Binary n-Cube Netz und das Baseline-Netz. Die **Abbildung 5.8** zeigt den Aufbau mehrstufiger dynamischer Netzwerke am Beispiel des Omega-Netzes. Darüber hinaus sind zum Vergleich das Indirect Binary n-Cube Netz (**Abbildung 5.9**) sowie das Baseline-Netz (**Abbildung 5.10**) dargestellt. An dieser Stelle besonders hervorzuheben sind die unterschiedlichen Abbildungen der Eingänge auf die Ausgänge trotz identischer Schalterstellungen in allen drei dargestellten Netzen. Dieser Effekt resultiert aus den unterschiedlichen Permutationen in den Verdrahtungsstufen zwischen den Schalterstufen. Das Omega-Netz besitzt 3 identische

[3] vgl. [Gr00, S. 62]

Shuffle-Permutationen σ_3. Das Indirect Binary n-Cube Netz besitzt zwei Butterfly-Permutationen β_2 und β_3 sowie eine Unshuffle-Permutation σ_3^{-1} am Ausgang. Das Baseline benötigt nur zwei Unshuffle-Permutationen σ_2^{-1} und σ_3^{-1}.

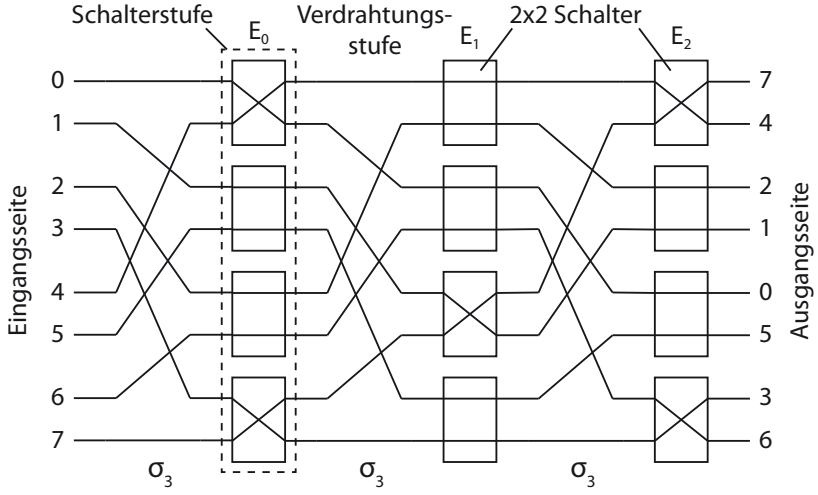


Abbildung 5.8: Omega-Netz

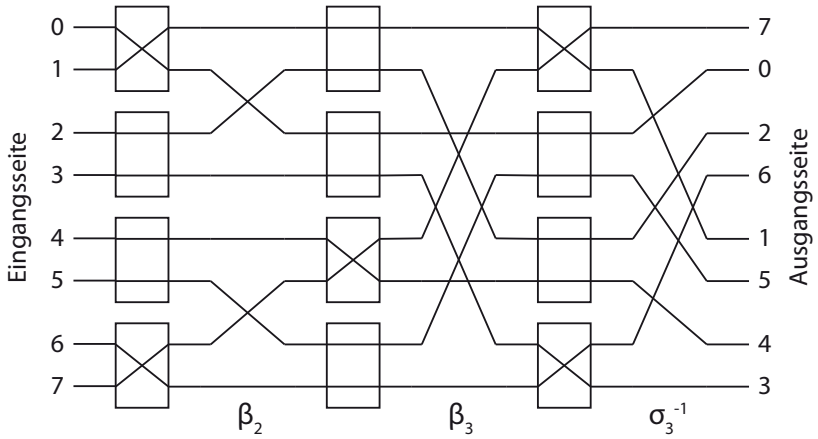


Abbildung 5.9: Indirect Binary n-Cube Netz

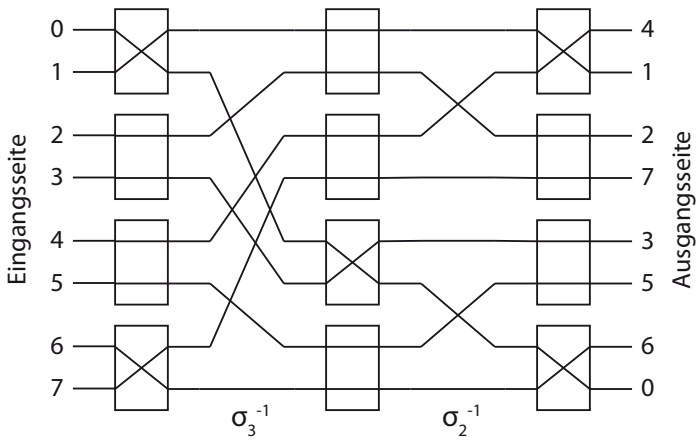


Abbildung 5.10: Baseline-Netz

5.3.2 Mathematische Beschreibung

Verkettete Permutationen. Aus mathematischer Sicht lassen sich mehrstufige Verbindungsnetzwerke als eine Verkettung linearer Permutationen beschreiben, welche im Ergebnis den Eingangsvektor auf den Ausgangsvektor abbildet. Unter einer Permutation versteht man die Veränderung der Anordnung einer endlichen Menge M von n Elementen durch Vertauschen ihrer Elemente. Verschiedene Anordnungen der Elemente ergeben sich aus den möglichen Permutationen. Die maximale Anzahl möglicher Permutationen P_n von n verschiedenen Elementen wird mit $P_n = n!$ angegeben [Br99, S. 743].

Bijektive Selbstabbildung. Eine Permutation von Elementen entsteht aus einer bijektiven Selbstabbildung der endlichen Menge M mit n Elementen in der folgenden Form:

$$p : M \rightarrow M \text{ mit } 0 \rightarrow p(0), 1 \rightarrow p(1), ..., (n-1) \rightarrow p(n-1) \qquad (5.4)$$

Matrizenschreibweise. Eine sehr übersichtliche und allgemeingültige Darstellung der eineindeutigen Relation zwischen den Elementen in einer Permutation bietet die Matrizenschreibweise [Ba97, S. 78]:

$$p = \begin{pmatrix} 0 & 1 & ... & (n-1) \\ p(0) & p(1) & ... & p(n-1) \end{pmatrix} \qquad (5.5)$$

Beispiel. Im Falle von mehrstufigen dynamischen Verbindungsnetzwerken besteht die Permutation aus der Abbildung von Adressen der Eingangsseite auf die Ausgangsseite des Netzes. Die Darstellung der Permutation von Eingängen auf Ausgänge würde für das Omega-Netz aus Abbildung 5.8 wie folgt aussehen:

$$p = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 7 & 4 & 2 & 1 & 0 & 5 & 3 & 6 \end{pmatrix} \qquad (5.6)$$

Zykelschreibweise. In der in Gleichung 5.6 dargestellten Matrizenschreibweise sind in der oberen Zeile die Adressen der Eingangsseite aufsteigend sortiert, links beginnend mit der kleinsten Eingangsadresse und der größten Eingangsadresse ganz rechts. In der unteren Zeile sind die Adressen der komplementären Ausgänge notiert. Unter der Voraussetzung, dass die Adressen der Eingänge fest definiert

(sortiert) sind, bietet die Zykelschreibweise eine sehr effiziente Form der Notation, da nicht benötigte Informationen einfach weggelassen werden können. Neben der Anordnung der Eingänge können im obigen Beispiel die Positionen 3 und 6 ebenfalls reduziert werden, da an diesen Stellen keine Veränderung erfolgt:

$$p = \begin{pmatrix} 0 & 7 & 6 & 3 & 1 & 4 \end{pmatrix} \tag{5.7}$$

Beispiel. Etwas unübersichtlicher wird diese Form der Notation in dem folgenden Beispiel. Hier kann die Permutation in Zykelschreibweise nur durch das Produkt disjunkter Zykel dargestellt werden. Die Reihenfolge der Verkettung spielt dabei keine Rolle [Be10, S. 176].

$$p = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 7 & 0 & 6 & 3 & 5 & 4 & 1 \end{pmatrix} \tag{5.8}$$

$$p = \begin{pmatrix} 0 & 2 \end{pmatrix} \circ \begin{pmatrix} 3 & 6 & 4 \end{pmatrix} \circ \begin{pmatrix} 1 & 7 \end{pmatrix} \tag{5.9}$$

Verkettung von Stufen. Wie in der Abbildung 5.8 auf Seite 82 am Beispiel des Omega-Netzes zu sehen ist, bestehen mehrstufige Netze aus mehreren Schalterstufen, die durch Verdrahtungsstufen miteinander verbunden sind. Dabei bildet jede Stufe eine Permutation der Eingänge auf die Ausgänge der Stufe. Verdrahtungsstufen erzeugen feste Permutationen, während die Permutationen der Schalterstufen dynamisch in Abhängigkeit der Schalterstellungen erzeugt werden. Die Verkettung aller Permutationen der Schalter- und Verdrahtungsstufen im Netz ergibt eine zyklische Permutation der Adressen der Eingangsseite auf die Ausgangsseite und umgekehrt. Die Topologie mehrstufiger Netze lässt sich mathematisch als Verkettung der einzelnen Stufen beschreiben. Im Beispiel des Omega-Netzes besteht die Topologie aus einer alternierenden Verkettung von Schalterstufen (E) und Shuffle-Verdrahtungsstufen (σ_n), beginnend mit einer Verdrahtungsstufe. Die mathematische Beschreibung des 3-stufigen Omega-Netzes aus Abbildung 5.8 mit 8 Ein- bzw. Ausgängen lautet somit:

$$\Omega_3 = \sigma_3 \circ E_0 \circ \sigma_3 \circ E_1 \circ \sigma_3 \circ E_2 \tag{5.10}$$

$$\Omega_3 = \left(\sigma_3 \circ E\right)^3 \tag{5.11}$$

Permutation der Verdrahtungsstufen. Die Aufgabe des Routings in mehrstufigen Netzen besteht darin, alle Kreuzschalter im Netz so einzustellen, dass durch die Verkettung aller Permutationen im Netz die gewünschte Abbildung der Eingänge auf die Ausgänge erreicht werden kann. Die Verdrahtungsstufen im Omega-Netz sind alle identisch und bestehen aus Shuffle-Permutationen dritter Ordnung:

$$\sigma_3 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 4 & 1 & 5 & 2 & 6 & 3 & 7 \end{pmatrix} \tag{5.12}$$

$$\sigma_3 = \begin{pmatrix} 1 & 4 & 2 \end{pmatrix} \circ \begin{pmatrix} 3 & 5 & 6 \end{pmatrix} \tag{5.13}$$

Permutation der Schalterstufen. Im Beispiel des Omega-Netzes aus Abbildung 5.8 lauten die Permutationen der einzelnen Schalterstufen:

$$E_0 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 0 & 2 & 3 & 4 & 5 & 7 & 6 \end{pmatrix} \tag{5.14}$$

$$E_1 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 1 & 2 & 3 & 5 & 4 & 6 & 7 \end{pmatrix} \tag{5.15}$$

$$E_2 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 0 & 2 & 3 & 4 & 5 & 7 & 6 \end{pmatrix} \tag{5.16}$$

bzw.

$$E_0 = \begin{pmatrix} 0 & 1 \end{pmatrix} \circ \begin{pmatrix} 6 & 7 \end{pmatrix} \tag{5.17}$$

$$E_1 = \begin{pmatrix} 4 & 5 \end{pmatrix} \tag{5.18}$$

$$E_2 = \begin{pmatrix} 0 & 1 \end{pmatrix} \circ \begin{pmatrix} 6 & 7 \end{pmatrix} \tag{5.19}$$

Für die Beschreibung von Schalterstufen eignet sich die Zykelschreibweise besonders, da Schalterstufen durch das Element der Kreuzschalter immer eine Komposition aus disjunkten 2er-Zykeln bilden. Zu beachten ist bei der Verkettung von Permutationen, dass diese von rechts nach links ausgewertet werden. Nehmen wir zum Beispiel die oberste Position in der Abbildung 5.8 auf Seite 82. Beginnend

mit der letzten Schalterstufe E_2 wird $0 \rightarrow 1$, $1 \rightarrow 4$, $4 \rightarrow 5$, $5 \rightarrow 6$, $6 \rightarrow 7$ und $7 \rightarrow 7$. Am obersten Ausgang wird also der Eingang 7 abgebildet. Genauso kann man beispielsweise mit der Position 6 verfahren: $6 \rightarrow 7$, $7 \rightarrow 7$, $7 \rightarrow 7$, $7 \rightarrow 7$, $7 \rightarrow 6$, $6 \rightarrow 3$. Am Ausgang 6 wird somit der Eingang 3 abgebildet. Umgekehrt funktioniert dieses Verfahren aufgrund der bijektiven Abbildung natürlich ebenfalls. Im einfachsten Fall, wenn alle Schalter in der Stellung *gerade* stehen, erhält man im Omega-Netz wie auch in den meisten anderen mehrstufigen Netzen die identische Anordnung der Eingänge auf die Ausgänge und es findet keine Vertauschung der Adressen statt.

5.3.3 Elementare Eigenschaften

Historische Entwicklung. Aufgrund ihrer langen historischen Entwicklung sind die elementaren Eigenschaften mehrstufiger Verbindungsnetzwerke, mit Ausnahme des Echtzeitverhaltens, heute weitgehend erforscht. Mehrstufige Netze wurden bereits in den 50er Jahren für die Leitungsvermittlung im Bereich der Telekommunikation eingesetzt, um zeitkritische Daten (z.B. einen Dialog im Telefongespräch) in einem sehr großen Netz zu übertragen. Zwei Jahrzehnte später wurden mehrstufige Netze aufgrund ihrer ausgezeichneten Skalierbarkeit und ihrer geringen Latenz bei der Datenübertragung in die Technik der Parallelrechner übernommen und weiterentwickelt[4]. In den bisher beschriebenen Techniken zeigen sich drei wesentliche Merkmale mehrstufiger Verbindungsnetzwerke, welche für die Interprozessorkommunikation in einem echtzeitfähigen MPSoC interessant sind.

Skalierbarkeit. Aufgrund ihrer skalierbaren Architektur stellen auch sehr große Teilnehmerzahlen für mehrstufige Netze kein topologisches Problem dar. Statische Netze besitzen zwar ebenfalls eine skalierbare Architektur, allerdings kann die Latenz der Nachrichtenübertragung in statischen Netzen stark variieren, da sie vom Abstand zwischen Sender- und Empfängerknoten, d.h. von der Anzahl der Zwischenknoten (*Multihops*), sowie von der Netzauslastung (*Hotspots*) abhängt. Bei mehrstufigen Netzen entspricht der Abstand von jedem Sender zu jedem Empfänger dagegen der Anzahl der durchlaufenen Stufen und ist deshalb konstant. Aufgrund des identischen Abstandes zwischen zwei beliebigen Sender- und Empfängerpaaren, spielt die Position von Sender und Empfänger in mehrstufigen Netzen keine Rolle für die Latenz der Nachricht.

[4] vgl. [Gr00, S. 96]

Paketvermittlung vs. Leitungsvermittlung. Statische Netze werden aufgrund ihrer topologischen Eigenschaften ausschließlich paketvermittelnd betrieben. Die zu übertragende Nachricht wird auf der Empfängerseite in kleine Pakete aufgeteilt. Gemäß dem verwendeten Übertragungsprotokoll werden jedem Paket zusätzliche Informationen, z.B. über den Sender und den Empfänger der Nachricht oder die Länge und die Art der Daten, angehängt. Dieses Vorgehen erzeugt für jedes Paket zusätzliche Daten neben der ursprünglichen Nachricht, die als Overhead bezeichnet werden. Anschließend wird jedes der Pakete über Zwischenknoten vom Sender zum Empfänger durch das Netz geleitet. Jeder Zwischenknoten muss das Paket öffnen und anhand der Zieladresse entsprechend seiner Routingstrategie in die Richtung des Empfängers weiterleiten. Auf der Empfängerseite müssen die Pakete in der richtigen Reihenfolge wieder zur urspünglichen Nachricht zusammengefügt werden. Im Gegensatz zu statischen Netzen können mehrstufige Netze auch leitungsvermittelnd betrieben werden. Der große Vorteil der Leitungsvermittlung besteht darin, dass eine direkte Verbindung vom jedem Sender zu jedem Empfänger aufgebaut werden kann. Sobald die Verbindung zwischen Sender und Empfänger hergestellt ist, werden die Daten ohne zusätzliche Verzögerung durch das Netz direkt vom Sender zum Empfänger der Nachricht übertragen. Dadurch wird die Latenz der Nachrichtenübermittlung unabhängig von der Netzgröße und auch unabhängig von der Position von Sender und Empfänger im Netz.

5.3.4 Echtzeit-Kategorien

Blockierungsfreiheit. Für die Übertragung von Echtzeitdaten spielt das deterministische Verhalten des Verbindungsnetzwerks eine entscheidende Rolle. Kollisionen, wie sie bei zufälligen gleichzeitigen Schreibzugriffen auf einem Datenbus oder bei besonders verkehrsbelasteten Routerknoten entstehen, müssen zugunsten eines deterministischen Verhaltens bei der Datenübertragung im Netz unbedingt vermieden werden. Mehrstufige Netze lassen sich hinsichtlich ihrer Verbindungseigenschaften in zwei Kategorien[5] einteilen: nicht-blockierungsfreie Netze (*blocking networks*) und blockierungsfreie Netze (*non-blocking* bzw. *rearrangeable networks*) (vgl. **Abbildung 5.11**). Zu den nicht-blockierungsfreien Netzen gehören alle Varianten der $\log_2 N$-Netze wie z.B. das Omega-, das Baseline- oder das Butterfly-Netz[6]. $\text{Log}_2 N$-Netze besitzen die gemeinsame Eigenschaft der Pfadein-

[5] vgl. Klassifizierung mehrstufiger Netzwerke in [Gr00, S. 100 ff.]

[6] vgl. [Ri97, S. 165 ff.]

deutigkeit, sind zueinander funktional äquivalent[7] und können jeden Eingang mit jedem freien Ausgang verbinden. Allerdings können nicht alle Permutationen der Eingänge auf die Ausgänge realisiert werden, da aufgrund der Pfadeindeutigkeit Kollisionen im Netz entstehen können. Das bedeutet, dass u.U. nicht alle Verbindungswünsche zur gleichen Zeit erfüllt werden können. Aus diesem Grund spricht man hier von nicht-blockierungsfreien Netzen. Die zweite Kategorie bilden die blockierungsfreien Netze. Diese Netze besitzen fast doppelt so viele Stufen wie nicht-blockierungsfreie Netze und können dadurch zu jeder Zeit jeden Eingang mit jedem freien Ausgang verbinden. Die Blockierungsfreiheit wird dadurch erreicht, dass alternative Pfade vom Sender zum Empfänger existieren, so dass bereits bestehende Kommunikationspfade bei Bedarf umgeordnet werden können.

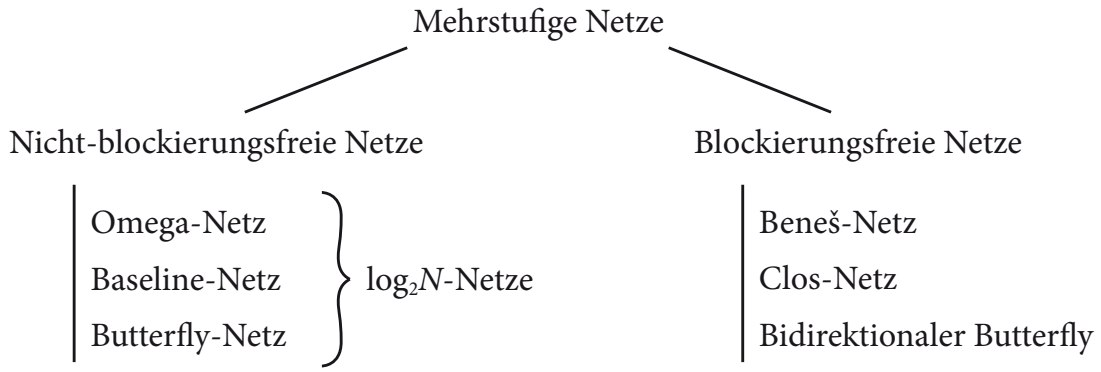


Abbildung 5.11: Echtzeit-Kategorien mehrstufiger dynamischer Netze

5.4 Nicht-blockierungsfreie Netze

Inhalt. In diesem Abschnitt wird das Echtzeitverhalten nicht-blockierungsfreier mehrstufiger dynamischer Netze untersucht. Zur Illustration dient das in [Wu80] eingeführte Baseline-Netz, einem Vertreter der $\log_2 N$-Netze, in der Größe 8x8.

5.4.1 Aufbau

Topologie der $\log_2 N$-Netze. Zu den bekanntesten Topologien nicht-blockierungsfreier Netze zählen die $\log_2 N$-Netze. Wie bereits im Abschnitt 5.3 ausgeführt wurde, bestehen $\log_2 N$-Netze aus in Stufen angeordneten Kreuzschaltern, die durch Permutationsfunktionen, den Verdrahtungsstufen, miteinander verbunden sind. Die Bezeichnung $\log_2 N$-Netz ergibt sich aus der Eigenschaft, dass sie aus n =

[7] Beweis der funktionalen Äquivalenz in [Wu80]

$\log_2 N$ Schalterstufen aufgebaut sind, wobei N die Anzahl der Ein- und Ausgänge im Netzwerk bezeichnet. Zu den $\log_2 N$-Netzen zählt auch das in der **Abbildung 5.12** dargestellte Baseline-Netz, welches erstmals in [Wu80] zum Beweis der topologischen und funktionalen Äquivalenz von $\log_2 N$-Netzen eingeführt wurde. Das Baseline-Netz besteht aus n Schalterstufen, welche durch n - 1 Unshuffle-Permutationen miteinander verbunden sind. Diese Konstruktion wird mathematisch durch die in Gleichung 5.20 beschriebene Verkettung von Permutationen beschrieben[8]:

$$BL_n = E_0 \circ \sigma_n^{-1} \circ E_1 \circ \sigma_{n-1}^{-1} \circ \ldots \circ \sigma_2^{-1} \circ E_{n-1} \qquad (5.20)$$

Der Vorteil des Baseline-Netzes. Das Baseline-Netz besteht wie das im Abschnitt 5.3.2 auf Seite 83 ff. beschriebene Omega-Netz aus n Schalterstufen, wobei jede Schalterstufe wiederum aus $N/2$ Kreuzschaltern besteht. Der topologische Unterschied zwischen beiden Netzen besteht in Anzahl und Art der Verdrahtungsstufen. Während das Omega-Netz wie auch das Indirect Binary n-Cube n Verdrahtungsstufen aufweist (vgl. Abbildung 5.8 und Abbildung 5.9, S. 82), benötigt das Baseline-Netz eine Verdrahtungsstufe weniger. Zudem weist das Baseline-Netz in den rechten Stufen eine zunehmend geringere Vermaschung auf, was der rekursiven Konstruktion des Baseline-Netzes[9] geschuldet ist, da es zur Ausgangsseite mit jeder Schalterstufe in doppelt so viele unabhängige Teilnetze zerfällt.

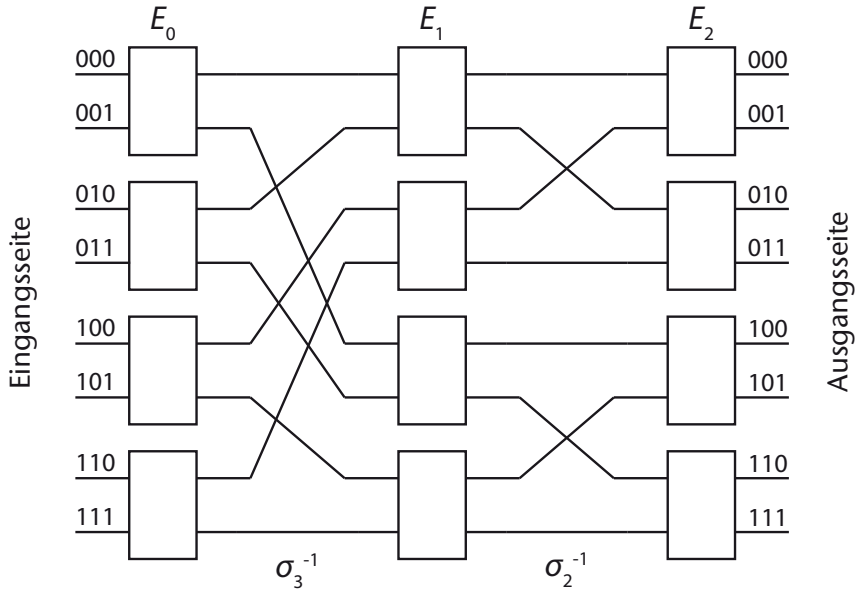


Abbildung 5.12: Mehrstufiges $\log_2 N$-Netz der Größe n = 3 (Baseline-Netz)

[8] vgl. [Ri97, S. 177]

[9] vgl. [Ri97, S. 177]

5.4.2 Routing

Einfacher Routingalgorithmus. Positiv zu werten ist, dass $\log_2 N$-Netze aufgrund ihrer Eigenschaft der Pfadeindeutigkeit einen äußerst einfachen Routingalgorithmus erlauben. Das Routing erfolgt einfach durch die bitweise Auswertung der binären Zieladresse in den einzelnen Schalterstufen. Dadurch kann der Routingalgorithmus als selbstorganisierendes Netzwerk auf der Ebene der Kreuzschalter implementiert werden und muss nicht, wie bei einem blockierungsfreien Netzwerk, in einer zentralen Instanz organisiert sein. In einem Baseline-Netz besteht zudem die Besonderheit, dass sich die Vermaschung, also die Zahl der möglichen Zieladressen, mit jeder Schalterstufe halbiert. Die sukzessive Auswertung der einzelnen Bits der binären Zieladresse erfolgt beginnend mit dem MSB in der ersten Schalterstufe bis zum LSB in der letzten Schalterstufe [Ri97, S. 179]. Steht in der Zieladresse eine *Null*, so wird der obere Schalterausgang genommen. Steht im anderen Fall in der Zieladresse eine *Eins*, so wird der untere Schalterausgang genommen. Die **Abbildung 5.13** zeigt das beschriebene Routing am Beispiel zweier Verbindungen. Die auszuwertenden Bits sind durch einen Pfeil kenntlich gemacht.

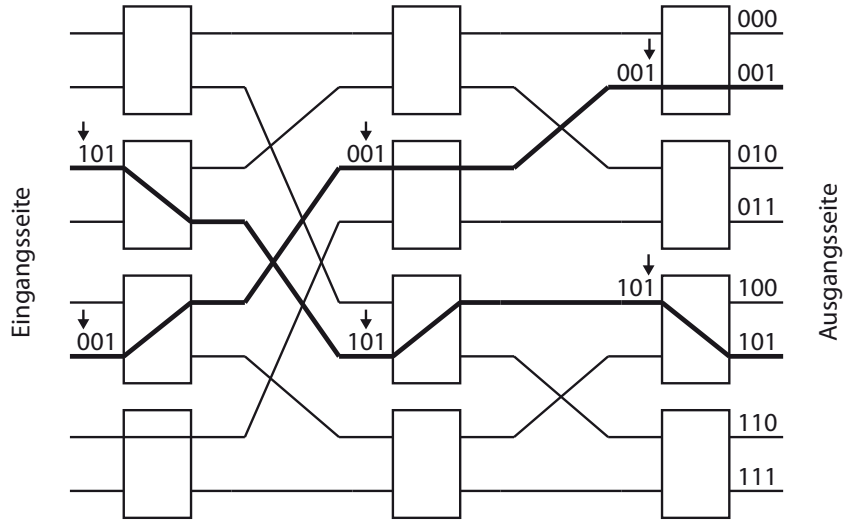


Abbildung 5.13: Routing im Baseline-Netz

Zeitgleiche Vermittlung mehrerer Verbindungen. Interessant für den Einsatz als on-Chip Lösung ist der relativ einfache Aufbau von $\log_2 N$-Netzen. Diese Netze benötigen die kleinste Anzahl von Schalter- und Verdrahtungsstufen, um jeden Eingang mit jedem Ausgang verbinden zu können. Zudem können in mehrstufigen Netzen mehrere Verbindungen zeitgleich vermittelt werden (vgl. Abbildung

5.13). Unter der Voraussetzung, dass jeder Sender zeitgleich an jeweils einen anderen Empfänger sendet, ist es möglich, mit einer Permutation der Eingänge auf die Ausgänge bis zu $N = 2^n$ Verbindungen gleichzeitig im Netz herzustellen. Das bedeutet, dass die maximale Übertragungsrate im $\log_2 N$-Netz mit der Anzahl N der Teilnehmer skaliert.

Einschränkungen. Trotz der prinzipiellen Möglichkeit, jeden Eingang mit jedem Ausgang verbinden zu können, gilt bei $\log_2 N$-Netzen die Einschränkung, dass nicht alle Permutationen der Eingänge auf die Ausgänge realisiert werden können. Die Ursache hierfür ist die Tatsache, dass bei $\log_2 N$-Netzen nur genau ein Pfad von jedem Eingang zu jedem Ausgang existiert. Dadurch kann es im Netz zu Ressourcenkonflikten in dem Fall kommen, wenn bei einer ungünstigen Konstellation der Verbindungswünsche zwei Verbindungen im Netz den gleichen Schalterausgang verwenden. Aufgrund der bitweisen Auswertung der binären Zieladresse müssen sich die beiden Zieladressen an beiden Schaltereingängen an der auszuwertenden Stelle unterscheiden, da sonst die Verbindung blockiert. Die **Abbildung 5.14** zeigt ein Beispiel, bei dem in der mittleren Stufe an beiden Eingängen des Schalters in der zweiten Ebene eine *Null* in den binären Zieladressen ausgewertet wird, wodurch sich die beiden Verbindungswünsche gegenseitig blockieren. Die dargestellte Permutation ist nicht realisierbar und somit *nicht gültig*. Aufgrund dieser Einschränkung werden $\log_2 N$-Netze auch als nicht-blockierungsfreie Netze (*blocking networks*) bezeichnet[10].

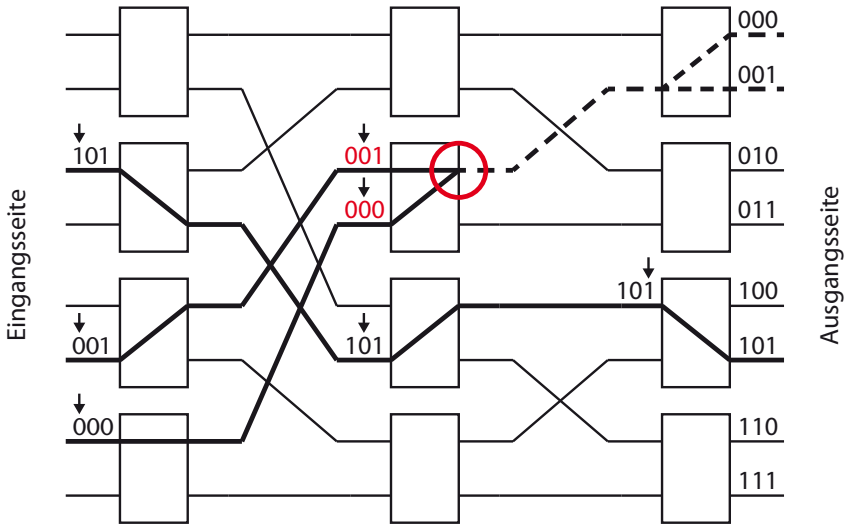


Abbildung 5.14: Blockierende Verbindung im Baseline-Netz

[10] vgl. [Ri97, S.165] und [Gr00, S. 106]

5.4.3 Nachrichten-Scheduling

Scheduling der Verbindungen. Um die Echtzeitfähigkeit von $\log_2 N$-Netzen trotz ihrer Nichtblockierungsfreiheit zu gewährleisten, muss das Vermitteln von Leitungen durch das Netz kontrolliert erfolgen. Dies geschieht mit Hilfe eines Schedulings der zu vermittelnden Verbindungen. Das Scheduling kann entweder statisch während der Konfigurationsphase oder dynamisch zur Laufzeit erfolgen.

Statisches Scheduling. Im statischen Fall können alle Verbindungswünsche in einer zentralen Schedulingtabelle festgehalten werden, die von einem Nachrichten-Scheduler zur Laufzeit abgearbeitet wird. So kann bereits vor der Laufzeit geprüft werden, ob alle Nachrichten fristgerecht zugestellt werden können. Die Voraussetzung hierfür ist allerdings, dass alle Nachrichten vor der Laufzeit bekannt sind.

Dynamisches Scheduling. Beim dynamischen Scheduling wird die Schedulingtabelle erst zur Laufzeit erstellt. Dies hat den Vorteil, dass das Nachrichten-Scheduling an die tatsächlich vorhandene Kommunikationslast angepasst werden kann. So können z.B. die Übertragungsrate bzw. die Taktfrequenz und damit der Energieverbrauch im Netz verringert werden, falls einzelne Funktionen in den Stand-by-Modus geschaltet werden und somit keine Nachrichten mehr senden bzw. empfangen. Die große Herausforderung bei dieser Lösung besteht darin, dass die Schedulingtabelle zur Laufzeit auf dem Chip generiert werden muss. Das Problem besteht in der erforderlichen Rechenleistung auf dem Chip, da das Erstellen der Schedulingtabelle - abhängig von der Scheduling-Methode, der Größe des Netzes und der Anzahl der Nachrichten plus deren Freiheitsgrade - relativ viel Rechenleistung respektive Rechenzeit benötigt. Zuletzt sei noch erwähnt, dass eine Aussage über das Zeitverhalten nur durch eine aufwändige Vorabsimulation der Interprozessorkommunikation oder durch eine Analyse zur Laufzeit möglich ist.

Kombiniertes Scheduling. Eine mögliche Alternative könnte darin bestehen, das statische und das dynamische Scheduling zu kombinieren. Das heißt, dass vor der Laufzeit mehrere Schedulingtabellen für verschiedene Szenarien erstellt werden, zwischen denen je nach Betriebszustand während der Laufzeit umgeschaltet wird (**Abbildung 5.15**). Der Vorteil dieser Kombination besteht darin, dass die Rechtzeitigkeit der Nachrichten noch vor der Laufzeit garantiert und gleichzeitig dynamisch auf unterschiedliche Kommunikationslasten reagiert werden kann.

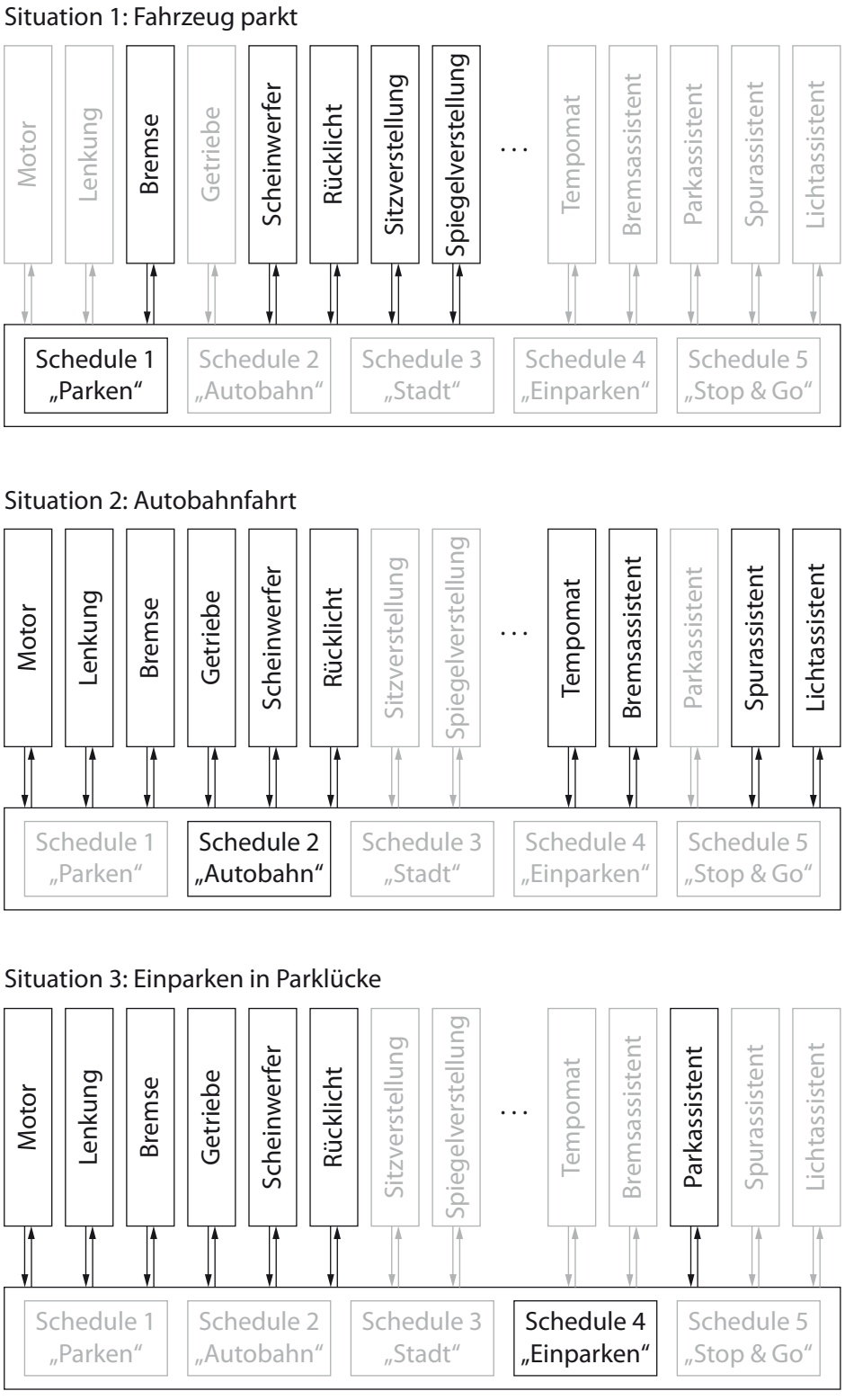


Abbildung 5.15: Situationsabhängiges Nachrichten-Scheduling (Theoretisches Beispiel)

5.4.3.1 Problembeschreibung

Gleichzeitige Übertragung. Wie in der Abbildung 5.14 zu sehen ist, können in mehrstufigen $\log_2 N$-Netzen zwar zeitgleich mehrere Nachrichtenübertragungen stattfinden, aber es sind nicht alle Kombinationen möglich. Das Problem entsteht durch Ressourcenkonflikte im Netz, die aufgrund belegter Pfade auftreten können. Im besten Fall können mit einer Permutation $p(\underline{I}) = \underline{O}$ gleichzeitig bis zu N Nachrichten übertragen werden können.

Konflikterkennung. Eine der Herausforderungen des Nachrichten-Schedulings in $\log_2 N$-Netzen besteht demzufolge darin, Nachrichten zu finden, die zeitgleich gesendet werden können. Dafür müssen die zu sendenden Nachrichten auf potentielle Ressourcenkonflikte untersucht werden[11]. Sind Nachrichten gefunden, die zueinander keine Konflikte aufweisen, so können diese Verbindungswünsche zu einer gültigen Permutation aufgebaut werden.

Rechtzeitigkeit der Nachrichten. Eine weitere Herausforderung besteht darin, alle Nachrichten innerhalb zeitlicher Grenzen zu senden, welche durch das Echtzeitsystem gegeben sind. Das heißt, alle Nachrichten müssen so zu gültigen Permutationen gebündelt werden, dass deren maximale Zustellzeiten eingehalten werden. Die Kernaufgabe des Nachrichtenschedulings besteht also darin, eine zeitliche Sequenz aus k gültigen Permutationen $\{p_0, p_1, \ldots, p_{k-1}\}$ zu finden, welche die gesamte Menge von m periodischen Nachrichten $\{e_0, e_1, \ldots, e_{m-1}\}$ abdeckt und dabei die Rechtzeitigkeit aller Nachrichten sicherstellt.

5.4.3.2 Konflikterkennung

Ressourcenkonflikte im Netz. Um blockierende Nachrichten im $\log_2 N$-Netz mittels eines Schedulers kontrolliert vermitteln zu können, müssen im ersten Schritt potentielle Ressourcenkonflikte im Netz erkannt werden, da miteinander konfligierende Nachrichten nicht gleichzeitig gesendet werden können. Solche Ressourcenkonflikte treten immer dann auf, wenn zwei Verbindungen denselben Schalterausgang verwenden. Die Aufgabe besteht also darin, die Nachrichtenpfade durch das Netz auf gemeinsam verwendete Schalterausgänge zu prüfen. Hierfür werden die Positionen der Pfade im Netz erfasst und auf Übereinstimmungen getestet[12].

[11] beschrieben in 5.4.3.2 Konflikterkennung, S. 94

[12] vgl. [Wu80]

Routing im Baseline-Netz. Die Gleichung 5.21 zeigt die Routing-Funktion durch das Baseline-Netz[13] der Größe $n = 3$. In dieser Darstellung ist der Weg durch das Netz klar nachzuvollziehen. Angefangen von der Position des Senders ($i_2 i_1 i_0$) auf der Eingangsseite bis zur Position des Empfängers ($o_2 o_1 o_0$) auf der Ausgangsseite sind alle Positionen im Netz beschrieben. Interessant sind neben den Positionen von Sender und Empfänger vor allem die Positionen am Ausgang der Schalterstufen, in diesem Fall die Bitfolgen ($i_2 i_1 o_2$) und ($o_2 i_2 o_1$). Stimmt eine der Bitfolgen überein, so existiert an dieser Stelle ein Konflikt im Netz.

$$\begin{aligned} I = i_2 i_1 i_0 &\xrightarrow{E(i_0,o_2)} i_2 i_1 o_2 \xrightarrow{\sigma_3^{-1}} \\ o_2 i_2 i_1 &\xrightarrow{E(i_1,o_1)} o_2 i_2 o_1 \xrightarrow{\sigma_2^{-1}} \\ o_2 o_1 i_2 &\xrightarrow{E(i_2,o_0)} o_2 o_1 o_0 = O \end{aligned} \qquad (5.21)$$

Die **Abbildung 5.16** zeigt die Routing-Funktion aus Gleichung 5.21 im Baseline-Netz am Beispiel von zwei Verbindungen, die keinen Konflikt aufweisen.

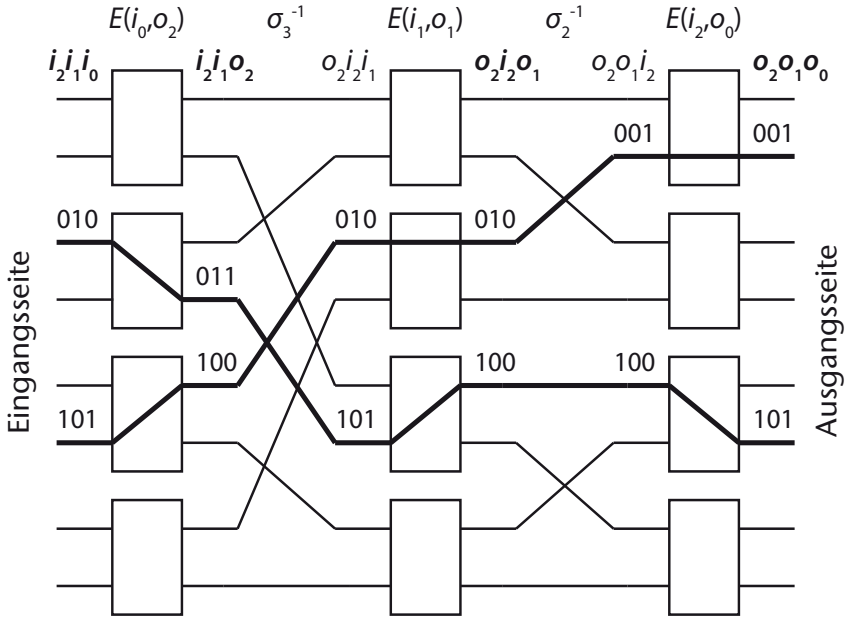


Abbildung 5.16: Routing-Funktion im Baseline-Netz

Konfliktttabelle. Als Beispiel für eine auftretende Konfliktsituation im Netz sollen die Verbindungswünsche aus Abbildung 5.14 auf Seite 91 dienen. Es soll nun geprüft werden, ob in diesem Beispiel irgendwelche Ressourcenkonflikte bestehen. Analog zu dem bereits in [Wu80] vorgestellten Verfahren werden alle Verbindungs-

[13] vgl. [Ri97, S. 179]

wünsche in der Konfliktttabelle aufgeführt. Wie in der **Tabelle 5.1** zu erkennen ist, besteht in Stufe 1 ein Ressourcenkonflikt zwischen den Verbindungen 5 → 1 und 6 → 0, da hier die Bitfolge am Ausgang der zweiten Schalterstufe übereinstimmt. Dieses Ergebnis deckt sich mit dem Pfadverlauf in Abbildung 5.14. Demzufolge können diese beiden Verbindungswünsche nicht zum gleichen Zeitpunkt vermittelt werden. Die Verbindung 2 → 5 weist hingegen keine Konflikte auf und kann somit jederzeit vermittelt werden.

Tabelle 5.1: Konfliktttabelle der Verbindungen (Beispiel)

Nr.	Sender → Empfänger	Eingang	Stufe 0	Stufe 1	Stufe 2
		$i_2 i_1 i_0$	$i_2 i_1 o_2$	$o_2 i_2 o_1$	$o_2 o_1 o_0$
1	2 → 5	010	011	100	101
2	5 → 1	101	100	**010**	001
3	6 → 0	110	110	**010**	000

5.4.3.3 Zeitverhalten

Periodische Nachrichten. Beim Nachrichten-Scheduling muss bereits vor der Laufzeit festgelegt werden, wann welche Nachricht übertragen wird. Das bedeutet, dass alle Nachrichten planbar sein müssen. Für periodische Nachrichten, welche in festen Zeitintervallen gesendet werden, stellt die Forderung kein Problem dar. Solche Nachrichten sind sehr gut planbar. In Steuer- und Regelsystemen stellen periodisch auftretende Nachrichten den Regelfall dar, da hier häufig Werte zyklisch aktualisiert werden müssen. So werden Nachrichten zwischen automobilen Steuergeräten i.d.R. in definierten Zyklen gesendet.

Aperiodische Nachrichten. Dagegen sind aperiodische Nachrichten, welche durch das Eintreten unvorherbestimmbarer Ereignisse ausgelöst werden, nur sehr schwer planbar. Eine Möglichkeit besteht darin, solche Nachrichten ebenfalls periodisch zu planen, wobei das maximal zulässige Zeitintervall durch die maximale Zustellzeit der Nachricht bzw. durch die maximale Antwortzeit des Rechensystems auf das Ereignis bestimmt wird. Dieses Vorgehen ist ähnlich dem technisch weit verbreiteten *Sensor-Polling*, bei dem ein Sensor zyklisch auf Änderungen getestet wird. Bei Eintreten eines Ereignisses wird die Nachricht zum nächsten geplanten Zeitpunkt gesendet. Im anderen Fall wird zum geplanten Zeitpunkt einfach keine Nachricht übertragen.

Tabelle 5.2: Zeitintervalle der Verbindungswünsche (Beispiel)

Nr.	Sender → Empfänger	Periodizität
1	2 → 5	10 ms
2	5 → 1	25 ms
3	6 → 0	10 ms

Periodizität. In der **Tabelle 5.2** sind als Beispiel die bereits aus der Abbildung 5.14 auf Seite 91 bekannten Verbindungswünsche gelistet. Neben den Informationen über Sender und Empfänger der Nachricht ist mit der Periodizität ein dritter Parameter aufgeführt, welcher das Zeitintervall zwischen zwei Übertragungen definiert. Wie bereits aus der Tabelle 5.1 bekannt ist, existiert ein Konflikt zwischen den Verbindungen Nr. 2 und 3. Der Schedule muss also derart gestaltet sein, dass diese beiden Nachrichten nie zum selben Zeitpunkt auftreten, während die Verbindung Nr. 1 davon unabhängig geplant werden kann.

Zeitliche Sequenz. Der Nachrichtenscheduler hat nun die Aufgabe, aus der Menge aller m Nachrichten $\{e_0, e_1, \ldots, e_{m-1}\}$ unter der Berücksichtigung von Ressourcenkonflikten (vgl. Tabelle 5.1) und Zeitbedingungen (vgl. Tabelle 5.2) verschiedene Teilmengen zu bilden und in gültige Permutationen zu gruppieren. Die aus den Verbindungswünschen entstandenen Permutationen werden in der Schedulingtabelle als zyklische Sequenz von k zeitlich aufeinanderfolgenden Permutationen $\{p_0, p_1, \ldots, p_{k-1}\}$ geplant.

Diskretisierung der Zeit. Da sich das Nachrichten-Scheduling auf einer kontinuierlichen Zeitachse bewegt, wird zunächst eine Einheitszeit T festgelegt, mit der die Zeitachse in diskrete Werte gerastert wird. Diese Einheitszeit orientiert sich an der Dauer einer Nachrichtenübertragung, welche von der normierten Länge der Nachrichten und der Übertragungsgeschwindigkeit im Netz abhängt. Je kürzer die Nachrichtenlänge und je größer die Übertragungsgeschwindigkeit, desto feiner wird das Raster und desto flexibler wird das Nachrichten-Scheduling.

5.4.3.4 Graphenmodell

Modellierung. Das Scheduling der Verbindungen im Netz stellt ein allgemeines Scheduling-Problem dar. Durch die Eigenschaft der blockierenden Verbindungen im nicht-blockierungsfreien Netz entstehen Konflikte zwischen den verschiedenen Nachrichten, welche z.B. durch ein Graphenmodell gelöst werden können.

Parameter. In der einfachsten Form enthält die Interprozessorkommunikation nur periodisch auftretende Nachrichten mit konstanter Länge. In diesem Fall wird jede Nachricht durch drei Parameter definiert: die *Nummer des Senders*, die *Nummer des Empfängers* und die *Periodizität der Nachricht*. Darüberhinaus sind auch Kommunikationsmodelle mit zusätzlichen Vorgaben wie variable Nachrichtenlängen, Prioritäten oder Jitter denkbar, die jedoch schnell zu einem deutlich komplexeren Scheduling-Modell führen. Aus diesem Grund soll nachfolgend nur der einfache Fall mit den drei genannten Parametern betrachtet werden.

Definition des Graphen. Aus der Menge der Nachrichten und den daraus entstehenden Konflikten im Netz wird ein Graph $G = (V, E)$ erstellt, wobei jede Nachricht einen Knoten $v \in V(G)$ im Graphen darstellt und für jeden Verbindungskonflikt im Netz eine Kante $e \in E(G)$ zwischen den betreffenden Knoten (bzw. Nachrichten) im Graphen eingezeichnet wird.

5.4.3.5 Konfliktlösung

Unterschiedliche Startzeitpunkte. Im Falle eines Ressourcenkonflikts muss das Nachrichten-Scheduling so gewählt sein, dass die miteinander konfligierenden Verbindungen nicht zum gleichen Zeitpunkt aufgebaut werden. Für das obige Beispiel müssen also zwei unterschiedliche Startzeitpunkte t_1 und t_2 für den Verbindungsaufbau gefunden werden. Ein entsprechendes Scheduling ist allerdings nur möglich, wenn die Perioden p_1 und p_2 der beiden Verbindungen einen gemeinsamen Teiler aufweisen. Ist dies nicht der Fall, so kann auch kein geeignetes Nachrichten-Scheduling gefunden werden.

Größter gemeinsamer Teiler. Um einen Ressourcenkonflikt im Netz aufzulösen, müssen die Startzeitpunkte t_1 und t_2 der beiden Nachrichten so gewählt werden, dass der Abstand der beiden Startzeitpunkte $|t_2\text{-}t_1|$ kein Vielfaches des größten gemeinsamen Teilers beider Perioden p_1 und p_2 (in diesem Beispiel 5 ms) beträgt. Dieser Sachverhalt ist in der Gleichung 5.22 noch einmal dargestellt. Im Fall von $ggT(p_1,p_2) == 1$ existiert allerdings keine Lösung.

$$|t_2 - t_1| \neq 0 \bmod ggT(p_1, p_2) \qquad (5.22)$$

Beispiel. Die Konfliktlösung soll kurz am Beispiel aus der Tabelle 5.2 erläutert werden. Angenommen, die Verbindung $5 \rightarrow 1$ soll mit einer Periodizität $p_1 = 25$ ms

und die Verbindung 6 → 0 soll mit einer Periodizität p_2 = 10 ms aufgebaut werden. Die Einheitszeit T im Netzwerk, welche gleichzeitig die maximale Übertragungsdauer sowie das Zeitraster für mögliche Startzeitpunkte darstellt, soll eine Millisekunde betragen. Der größte gemeinsame Teiler der Perioden p_1 und p_2 beträgt $ggT(p_1, p_2)$ = 5 ms. Der Startzeitpunkt der Verbindung 5 → 1 soll $t_1 = 0$ *ms* betragen. Der Startzeitpunkt t_2 für die Verbindung 6 → 0 muss folglich in einem Bereich von {0,…, p_2-1} ms liegen. Mögliche Startzeitpunkte für t_2 sind somit 1, 2, 3, 4, 6, 7, 8 oder 9 ms (vgl. **Abbildung 5.17**).

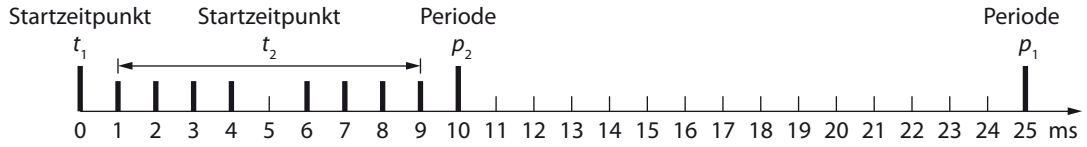


Abbildung 5.17: Mögliche Startzeitpunkte im Nachrichten-Scheduling

5.4.4 Fazit

Vorteile. Nicht-blockierungsfreie Netze in der Form der $\log_2 N$-Netze bieten aufgrund ihrer topologischen Eigenschaften eine hervorragende Skalierbarkeit. Die maximale Übertragungsrate des Netzes steigt proportional mit der Anzahl N der Teilnehmer, während die Kosten in Hardware mit $N/2 \log_2 N$ Kreuzschaltern relativ gering ausfallen. Zudem bieten $\log_2 N$-Netze einen sehr einfachen Routing-Algorithmus, welcher dezentral organisiert auf der Ebene der Schalter implementiert werden kann. Dies ist besonders bei sehr großen Netzen von Vorteil, da keine zentrale Routinginstanz erforderlich ist.

Einschränkungen. Der große Nachteil nicht-blockierungsfreier Netze liegt darin, dass aufgrund möglicher Ressourcenkonflikte u.U. nicht jede Verbindung zu jedem Zeitpunkt aufgebaut werden kann. Aus diesem Grund ist ein Scheduling der Verbindungen nötig, welches aufgrund der erforderlichen Rechenleistung bereits vor der Laufzeit erfolgen muss. Positiv zu bewerten ist allerdings, dass potentielle Ressourcenkonflikte im Netz leicht zu detektieren sind.

Echtzeitfähgkeit. Das Nachrichten-Scheduling bietet bereits vor der Laufzeit die Möglichkeit, Aussagen über das Zeitverhalten der Nachrichtenübertragung im Netz zu treffen. Unter dieser Voraussetzung sind nicht-blockierungsfreie Netze für die Interprozessorkommunikation im Echtzeitparallelrechner geeignet.

5.5 Blockierungsfreie Netze

Inhalt. In diesem Abschnitt wird das Echtzeitverhalten blockierungsfreier mehrstufiger dynamischer Netze am Beispiel des Beneš-Netzes untersucht. Neben den Grundlagen zum Aufbau und der Funktion blockierungsfreier Netze wird gezeigt, ob und wie sich bekannte Routing-Verfahren für die Interprozessorkommunikation im Echtzeitparallelrechner eignen.

5.5.1 Aufbau

Verdopplung der Stufenzahl. Schaltet man zwei nicht-blockierungsfreie $\log_2 N$-Netze hintereinander, so erhält man ein Netz der zweiten Echtzeit-Kategorie[14] mehrstufiger Netze, der blockierungsfreien Netze. Die **Abbildung 5.18** zeigt den Aufbau des Beneš-Netzes[15] als einen Vertreter der *Rearrangeable Networks*[16] mit konfliktfreiem Routing der Beispielverbindungen aus Abbildung 5.14. Das Beneš-Netz besteht aus der Verkettung von zwei spiegelbildlichen Baseline-Netzen, wobei die letzte Stufe des ersten Netzes und die erste Stufe des zweiten Netzes aufgrund ihrer Redundanz miteinander verschmelzen [Ri97, S. 240].

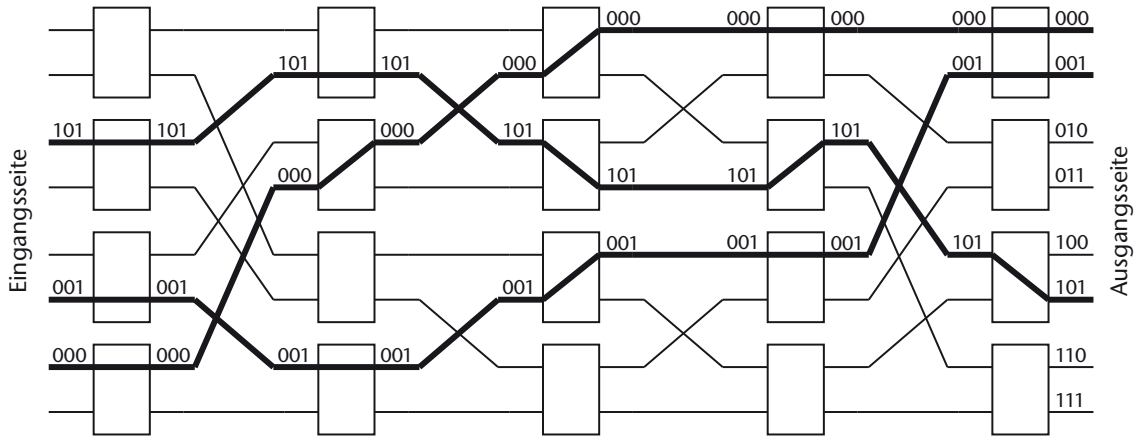


Abbildung 5.18: Blockierungsfreie Verbindung im Beneš-Netz

Alternative Pfade. Durch die auf $2\log_2 N-1$ vergrößerte Stufenanzahl besitzt das Beneš-Netz die Eigenschaft, zu jedem Zeitpunkt jeden Eingang mit jedem freien Ausgang verbinden zu können. Diese für Echtzeitkommunikation wichtige Eigenschaft basiert auf der Existenz von $N/2$ alternativen Pfaden in dem fast doppelt

[14] vgl. Abbildung 5.11: Echtzeit-Kategorien mehrstufiger dynamischer Netze, S. 88

[15] erstmals beschrieben im Jahr 1962 durch Václav E. Beneš in [Be62]

[16] vgl. Klassifikation leitungsvermittelnder mehrstufiger Netze in [Gr00, S. 101]

so großen Netz. Der Klassenname *Rearrangeable Network* rührt daher, dass im Konfliktfall bestehende Verbindungen auf alternative Pfade verlegt werden können. Dadurch können Konfliktsituationen wie in der Abbildung 5.14 auf Seite 91 gelöst werden. In der **Abbildung 5.19** sind im Beneš-Netz der Größe 8 x 8 die $N/2 = 4$ möglichen Pfade für die Beispielverbindung $2 \rightarrow 5$ dargestellt. Die Anzahl der möglichen Pfade von jedem Eingang zu jedem Ausgang steigt mit $N/2$ proportional mit der Anzahl der Ein- und Ausgänge.

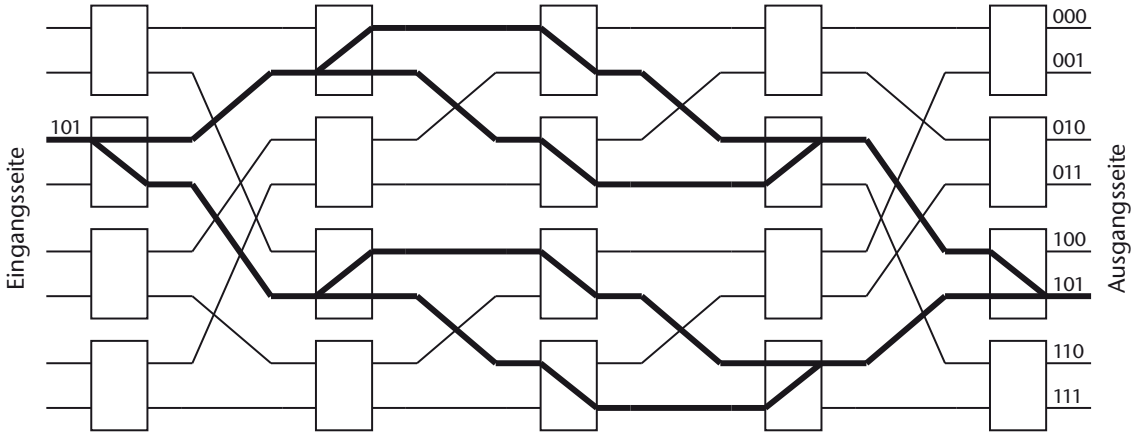


Abbildung 5.19: Alternative Pfade im blockierungsfreien Beneš-Netz der Größe 8 x 8

Komplexes Routing. Das Beneš-Netz bietet gegenüber den nicht-blockierungsfreien $\log_2 N$-Netzen den großen Vorteil, dass alle möglichen Permutationen der Eingänge auf die Ausgänge konfliktfrei geroutet werden können. Nachteilig ist hier allerdings, dass das Routing aufgrund der $N/2$ alternativen Pfade und der damit fehlenden Pfadeindeutigkeit deutlich schwieriger zu realisieren ist als bei den pfadeindeutigen, dafür aber nicht-blockierungsfreien $\log_2 N$-Netzen[17]. Zudem müssen bei jedem Aufbau einer neuen Verbindung möglicherweise bereits bestehende Verbindungen von anderen Sender-/Empfängerpaaren auf alternative Netzpfade gelegt werden, so dass für jede neue Verbindung praktisch das komplette Netz neu geroutet werden muss. Da dieser Vorgang bei sequentieller Ausführung eine nichtlineare Zeitkomplexität besitzt[18], entstehen hier im Vergleich zu den nicht-blockierungsfreien $\log_2 N$-Netzen neben den zusätzlichen Stufen weitere Kosten in der Zeit wie auch in der Hardware durch das recht aufwändige Routing. Aus diesem Grund soll in den nachfolgenden Abschnitten das Routingverfahren als notwendiges Kriterium im Mittelpunkt der Betrachtungen stehen.

[17] vgl. 5.4.2 Routing, S. 90

[18] vgl. Abbildung 5.27: Anzahl der Routingschritte in Abhängigkeit von der Netzgröße, S. 110

5.5.2 Dezentrales Routing

Routing auf Schalterebene. Die schnellste Möglichkeit, ein mehrstufiges Netz zu routen, besteht darin, alle Schalterstellungen im Netz simultan festzulegen. Im Jahr 1981 stellten Nassimi und Sahni deshalb ihre Problemlösung in Form eines selbstorganisierenden Beneš-Netzes vor [Na81]. Dieser Ansatz basiert auf der Idee, jedem Kreuzschalter im Netz eine möglichst einfache Logik für das Routing der gewünschten Verbindung zu implementieren. Wie in einem $\log_2 N$-Netz sollte jeder Schalter in der Lage sein, anhand der Zieladresse der zu routenden Verbindung am Schaltereingang selbst die richtige Schalterstellung zu ermitteln. Auf diese Weise können alle Schalterstellungen im Netz zeitgleich und unabhängig voneinander gesetzt werden.

Beispiel. Die **Abbildung 5.20** zeigt das auf Schalterebene organisierte Routing von Nassimi und Sahni am Beispiel einer Bit-Reversal Permutation der Eingänge auf die Ausgänge. Das dargestellte Routingverfahren ist äußerst simpel und entspricht im Prinzip dem Routing von $\log_2 N$-Netzen[19]. Entscheidend für die Schalterstellung ist jeweils ein Bit in der binären Zieladresse am oberen Eingang des Schalters. Welche Bitposition ausgewertet wird, ergibt sich aus der Position der Schalterstufe im Netz. Die entsprechenden Bitpositionen sind in dem dargestellten Beispiel mit Pfeilen gekennzeichnet. Steht an der Position in der oberen Zieladresse eine *Null*, so wird der obere Ausgang genommen (Schalterstellung = *gerade*). Im Umkehrschluss wird bei einer *Eins* in der oberen Zieladresse der untere Ausgang genommen (Schalterstellung = *gekreuzt*).

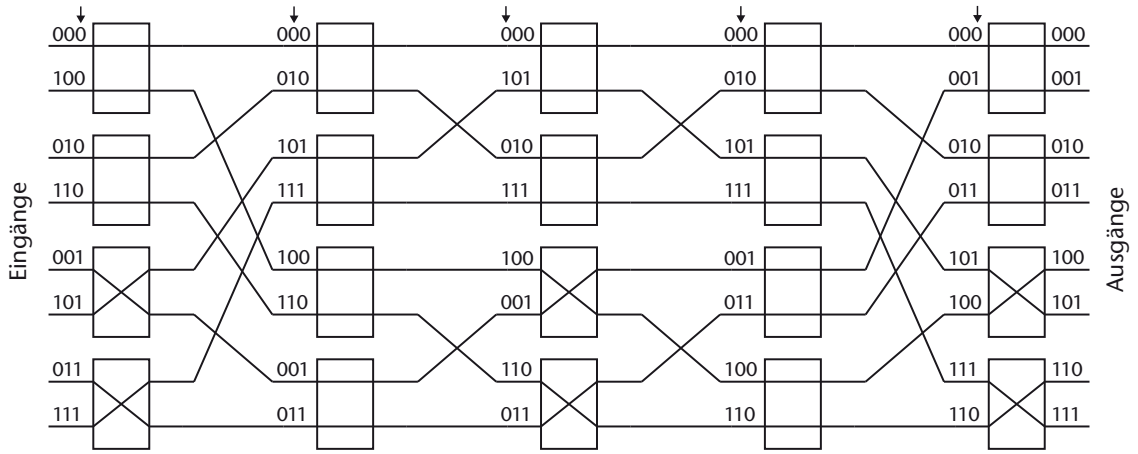


Abbildung 5.20: Erfolgreiches Routing im Beneš-Netz nach Nassimi und Sahni [Na81]

[19] vgl. 5.4.2 Routing, S. 90

Einschränkungen im Routing. Leider unterliegt der von Nassimi und Sahni vorgestellte Algorithmus starken Einschränkungen bei der Zahl der darstellbaren Permutationen. So funktioniert das Routing neben der gezeigten Bit-Reversal Permutation nur bei einigen wenigen Permutationen wie Perfect Shuffle, Unshuffle oder der Matrix-Transposition. Die für die Steuergerätekommunikation notwendige Forderung zum Routen der Menge aller *N*! möglichen Permutationen kann dieses Routingverfahren leider nicht erfüllen.

Weiterentwicklung. Aufgrund der stark eingeschränkten Routingfunktionalität des in [Na81] vorgestellten Verfahrens verfolgten Raghavendra und Boppana die Idee von Nassimi und Sahni weiter und veröffentlichten im Jahr 1991 eine weiterentwickelte Variante des dezentralen Routingverfahrens [Ra91]. Das Ziel dieser Weiterentwicklung bestand darin, die Klasse der linearen Permutationen[20] im Beneš-Netz zu routen. Als Beispiel verwendeten sie die in der **Abbildung 5.21** dargestellte Permutation, welche durch das Routingverfahren von Nassimi und Sahni nicht erfolgreich geroutet werden konnte. Das Grundprinzip des Routingverfahrens ist in der Weiterentwicklung durch Raghavendra und Boppana gleich geblieben. So wird die Schalterstellung ebenfalls bitweise aus der binären Form der Zieladresse am Schaltereingang abgeleitet. Auch die Abhängigkeit der Bitposition von der Position der Schalterstufe im Netz ist identisch. Der Unterschied besteht nun darin, welche der beiden Eingangsadressen ausgewertet wird. Während bei Nassimi und Sahni in allen Stufen die einzelnen Bits der oberen binären Eingangsadresse auf 0 oder 1 getestet wurden, erfolgt nun in den ersten (n-1) Stufen des Netzes die Bit-Auswertung in der wertemäßig kleineren der beiden binären Eingangsadressen. Steht in der wertemäßig kleineren Adresse eine *Null*, so wird für diesen Eingang der obere Ausgang genommen. Im Umkehrschluss wird bei einer *Eins* am Eingang der untere Ausgang genommen. Auf diese Weise ergibt sich das Routingschema wie in der **Abbildung 5.22** dargestellt. Das Routing-Bit ist in der Darstellung wieder durch einen kleinen Pfeil gekennzeichnet.

Einschränkungen. Eigene Untersuchungen zum Routingverfahren von Raghavendra und Boppana haben allerdings ergeben, dass auch dieses Verfahren nicht die bei der Interprozessorkommunikation erforderlichen *N*! möglichen Permutationen erfolgreich routen kann. So zeigt die **Abbildung 5.23** eine Permutation, bei der das Routingverfahren aus [Ra91] nicht funktioniert.

[20] vgl. [Ra90]

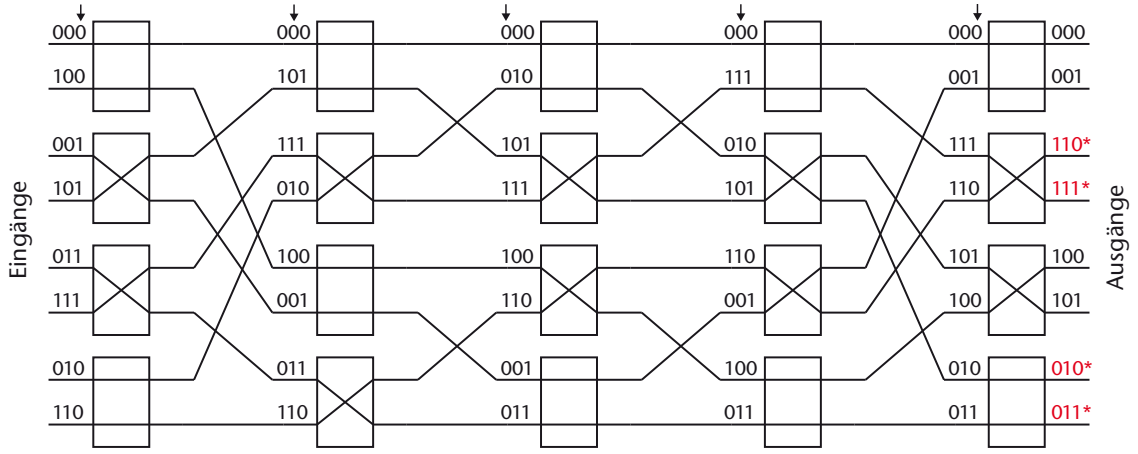


Abbildung 5.21: Erfolgloses Routing im Beneš-Netz nach Nassimi und Sahni [Na81]

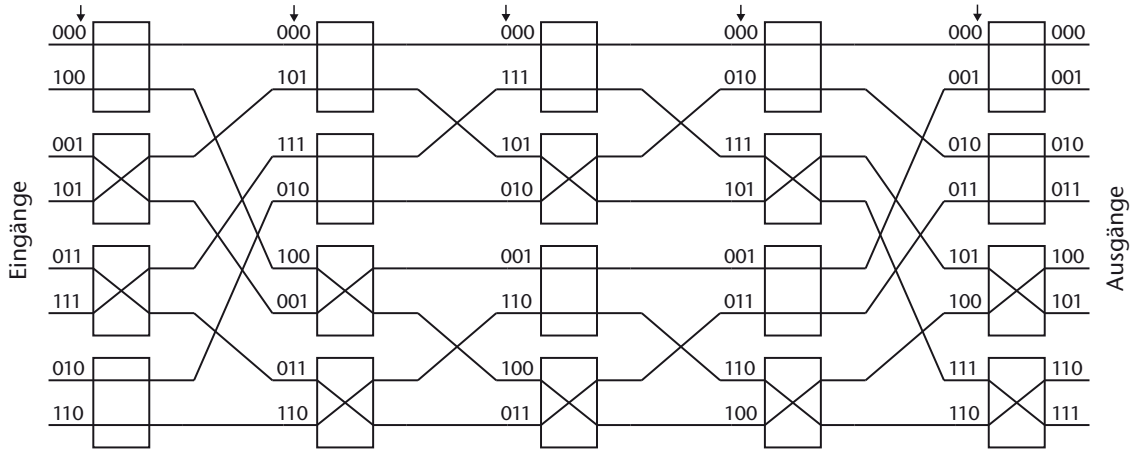


Abbildung 5.22: Erfolgreiches Routing im Beneš-Netz nach Raghavendra und Boppana [Ra91]

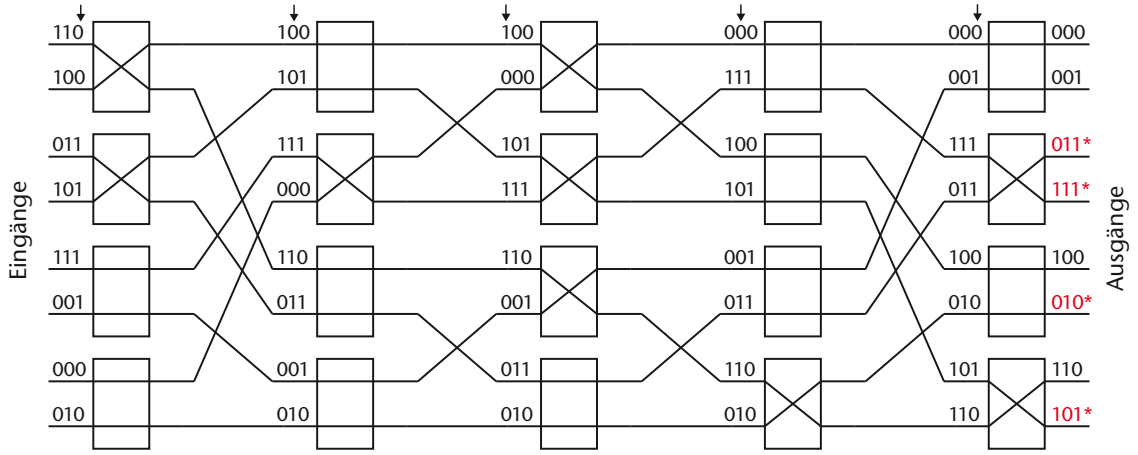


Abbildung 5.23: Erfolgloses Routing im Beneš-Netz nach Raghavendra und Boppana [Ra91]

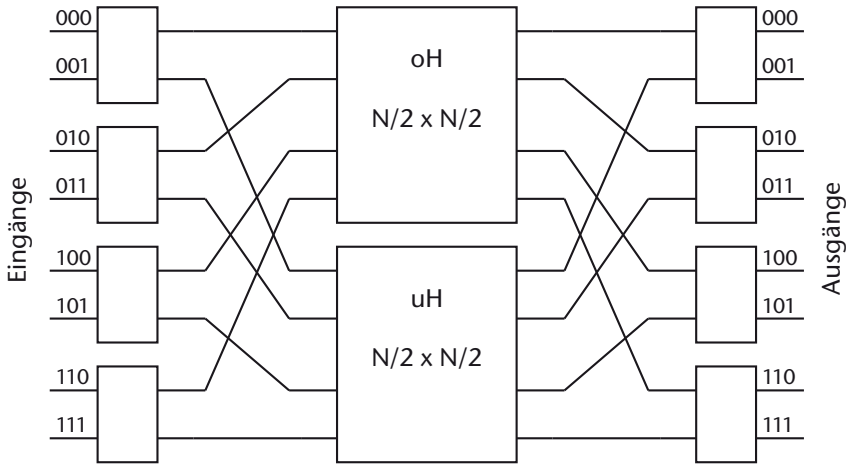


Abbildung 5.24: Rekursiver Aufbau des Beneš-Netzes[21]

Problem beim dezentralen Routing. Um das Problem beim dezentralen Routingverfahren im Beneš-Netz zu erläutern, ist in der **Abbildung 5.24** der Aufbau der mittleren Stufen durch Teilnetze abstrahiert dargestellt. Rechts und links davon befinden sich die erste und die letzte Stufe des Netzes. Das Beneš-Netz ist symmetrisch aus zwei spiegelbildlichen Baseline-Netzen zusammengesetzt, wobei die letzte Stufe des ersten (linken) Baseline-Netzes und die erste Stufe des zweiten (rechten) inversen Baseline-Netzes redundant in einer Mittelstufe aufgehen. Wie in der Abbildung 5.24 zu erkennen ist, teilt sich das Beneš-Netz zur Mitte in zwei unabhängige Teilnetze: einer oberen Hälfte (oH) und einer unteren Hälfte (uH). Aufgrund der ersten Unshuffle-Verdrahtungsstufe sind die oberen Ausgänge der Schalter der ersten Schalterstufe mit der oberen Hälfte und die unteren Ausgänge der Schalter mit der unteren Hälfte verbunden. Analog dazu sind aufgrund der letzten Shuffle-Verdrahtungsstufe die Ausgänge der oberen Hälfte mit den oberen Eingängen und die Ausgänge der unteren Hälfte mit den unteren Eingängen der Schalter der letzten Schalterstufe verbunden. Daraus folgt, dass die zu routenden Pfade nicht unabhängig voneinander betrachtet werden können. In der ersten Schalterstufe müssen die Schalter so eingestellt sein, dass die komplementären Ausgänge jedes Schalters der letzten Schalterstufe in verschiedene Teilnetze geroutet werden. So scheitert das Routing in der Abbildung 5.23 unter anderem, weil die komplementären Zieladressen 010 und 011 des zweiten Schalters der letzten Stufe in der ersten Schalterstufe beide in das untere Teilnetz geroutet werden. Dadurch kann höchstens eine der beiden Zieladressen, oder wie in diesem Fall auch keine von beiden, bis zum gewünschten Ausgang geroutet werden.

[21] grafische Darstellung aus [Gr00, S. 123]

Fazit. Das nicht-blockierungsfreie Baseline-Netz bildet die topologische Basis für den Aufbau des blockierungsfreien Beneš-Netzes. Durch die Erweiterung der Anzahl der Stufen auf fast das Doppelte entstehen $N/2$ alternative Pfade im Netz, so dass durch Umordnen bestehender Verbindungen zu jedem Zeitpunkt jeder Eingang mit jedem freien Ausgang verbunden werden kann. Das Problem besteht hier im Routing-Verfahren. Der rekursive Aufbau des Baseline-Netzes erzeugt in den inneren Stufen des Beneš-Netzes voneinander unabhängige Teilnetze. Dies führt wiederum dazu, dass die beiden komplementären Ausgänge der letzten Schalterstufe zwingend voneinander abhängen, da diese in verschiedene Teilnetze geroutet werden. Im Gegensatz zu den $\log_2 N$-Netzen ist dadurch im Beneš-Netz ein dezentrales selbstorganisierendes Routing auf Schalterebene nicht möglich.

5.5.3 Looping-Routing

Grundlagen. Ein lange bekanntes Routing-Verfahren für das Beneš-Netz ist das *Looping-* bzw. *Schleifen-Routing*[22]. Der Name verdeutlicht bereits das sequentielle Prinzip dieses Verfahrens. Im Schleifenverfahren werden die Schalter im Netz nach dem Ursache/Wirkung-Prinzip gesetzt, bis das gesamte Netz geroutet ist. Das Setzen der Schalter erfolgt dabei nach Stufen geordnet von außen nach innen[23]. Die nachfolgend verwendeten Schalterbezeichnungen sind in der **Abbildung 5.25** dargestellt. Da die Schalter in Stufen und Ebenen angeordnet sind, wird in der Darstellung für jeden Schalter die Bezeichnung *Stufe.Ebene* verwendet.

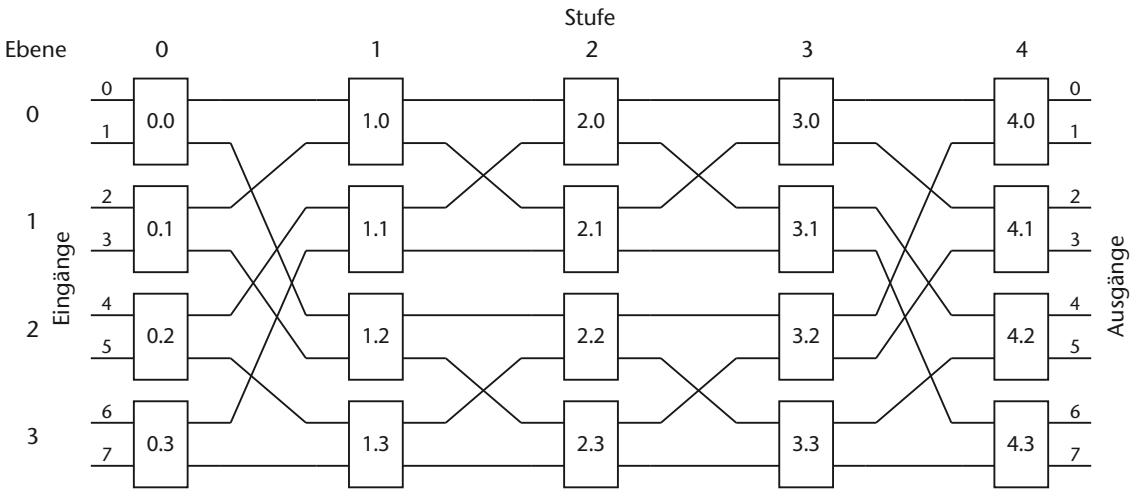


Abbildung 5.25: Schalterbezeichnungen im Beneš-Netz

[22] Dekomposition durch Schleifen (*decomposition by looping*), [Op71, S. 1588]

[23] vgl. [Ri97, S. 242 ff.]

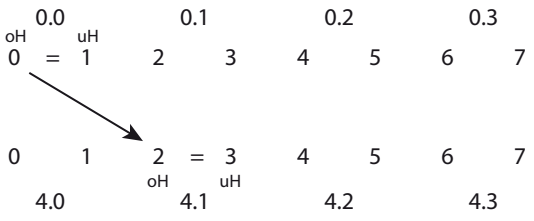
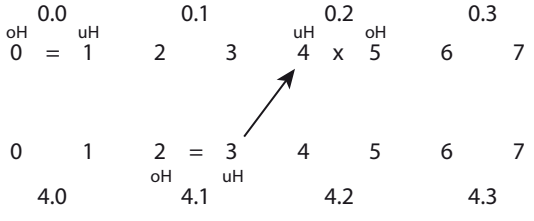
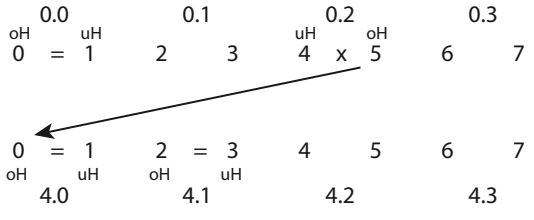
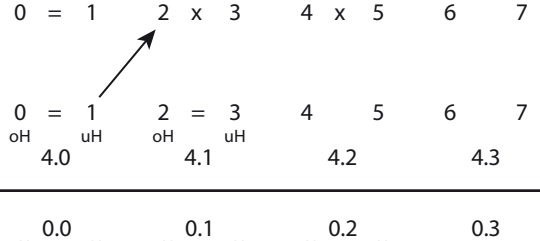
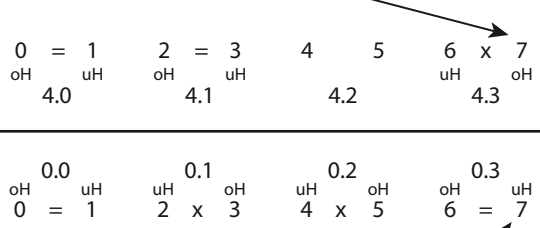
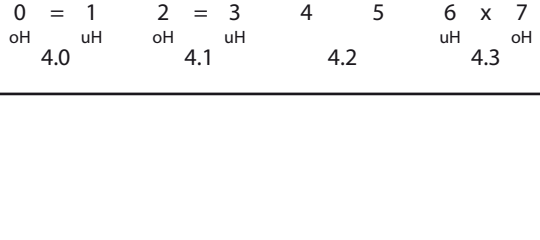
Beispiel. Das Prinzip des Looping-Routing lässt sich am besten an einem Beispiel erklären. Gegeben sei die Permutation p mit

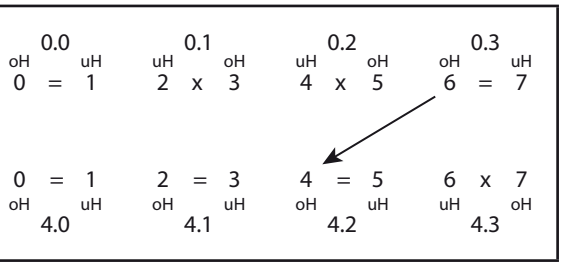
$$p = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 5 & 1 & 7 & 3 & 0 & 4 & 6 \end{pmatrix} \qquad (5.23)$$

in einem Beneš-Netz der Größe 8 x 8 (vgl. Abbildung 5.25). Das Looping-Routing beginnt nun mit dem Setzen der Schalter in den beiden äußeren Stufen, also den Schaltern mit den Bezeichnungen 0.x und 4.x in den Stufen 0 und 4. Angefangen mit dem ersten Schalter 0.0 der ersten Stufe, pendelt das Verfahren zwischen den beiden äußeren Stufen immer hin und her, bis alle Schalter der beiden äußeren Stufen gesetzt sind. Anschließend wird der Vorgang analog bei den inneren Stufen 1 und 3 und anschließend der mittleren Stufe 2 fortgesetzt, bis alle Schalterstellungen bestimmt sind. Der Hintergrund dieser Vorgehensweise besteht in dem rekursiven Aufbau der beiden spiegelbildlichen Baseline-Topologien im Beneš-Netz. Durch die Halbierung der erreichbaren Adressen zur Mitte des Beneš-Netzes wird je ein Pfad von jedem Schalter zur inneren Stufe in die obere Hälfte und der andere Pfad zwangsweise in die untere Hälfte geroutet (vgl. Abbildung 5.24). Dadurch entsteht eine Abhängigkeit zwischen den beiden komplementären Adressen eines jeden Schalters.

Vorgehensweise. Die **Tabelle 5.3** beschreibt detailliert die Teilschritte für das Looping-Routing der äußeren Stufen am Beispiel aus der Gleichung 5.23. Im ersten Schritt wird der Schalter 0.0 der Stufe 0 auf Stellung *gerade* (=) gesetzt. Daraus resultiert, dass der Eingang 0 sowie der Ausgang der korrespondierenden Zieladresse (hier Ausgang 2) in die obere Hälfte (oH) der beiden inneren Stufen 1 und 3 geroutet werden. Dadurch wird der Schalter 4.1 der Stufe 4 ebenfalls auf *gerade* gesetzt. Der komplementäre Ausgang (hier Ausgang 3) wird zwangsläufig in die untere Hälfte (uH) geroutet. Daraus folgt wiederum, dass der korrespondierende Eingang dieser Zieladresse (hier Eingang 4) ebenfalls in die untere Hälfte geroutet werden muss. Aus diesem Grund wird der Schalter 0.2 der Stufe 0 auf die Stellung *gekreuzt* (x) gesetzt. Dieses Verfahren pendelt solange zwischen den Stufen 0 und 4 hin und her, bis alle Schalterstellungen in den beiden äußeren Stufen bestimmt sind. Im Anschluss setzt sich das Verfahren in den beiden inneren Stufen 1 und 3 in gleicher Weise fort, bis das gesamte Netz, inklusive der Mittelstufe, geroutet ist.

Tabelle 5.3: Looping-Routing der beiden äußeren Stufen im Beneš-Netz

Schritt 1: Der Schalter 0.0 wird willkürlich auf „=“ gesetzt, wodurch der Eingang 0 in die obere Hälfte (oH) geroutet wird. Für Schalter 4.1 ergibt sich ebenfalls die Stellung „=“, da der korrespondierende Ausgang 2 ebenfalls in die obere Hälfte geroutet werden muss.	
Schritt 2: Aus der Stellung des Schalters 4.1 ergibt sich die Stellung von Schalter 0.2, da der mit Ausgang 3 korrespondierende Eingang 4 ebenfalls in die untere Hälfte (uH) geroutet werden muss.	
Schritt 3: Der Schalter 4.0 wird auf Stellung „=“ gesetzt, da der mit Eingang 5 korrespondierende Ausgang 0 in die obere Hälfte (oH) geroutet werden muss.	
Schritt 4: Der Schalter 0.1 wird auf Stellung „x“ gesetzt, da der mit Ausgang 1 korrespondierende Eingang 2 in die untere Hälfte (uH) geroutet werden muss.	
Schritt 5: Der Schalter 4.3 wird auf Stellung „x“ gesetzt, da der mit Eingang 3 korrespondierende Ausgang 7 in die obere Hälfte (oH) geroutet werden muss.	
Schritt 6: Der Schalter 0.3 wird auf Stellung „=“ gesetzt, da der mit Ausgang 6 korrespondierende Eingang 7 in die untere Hälfte (uH) geroutet werden muss.	

Schritt 7: Der Schalter 4.2 wird auf Stellung „=“ gesetzt, da der mit Eingang 6 korrespondierende Ausgang 4 in die obere Hälfte (oH) geroutet werden muss.	

Die in der Tabelle 5.3 dargestellten 7 Teilschritte reichen aus, um im Beispiel alle Schalterstellungen in den äußeren Stufen 0 und 4 zu bestimmen. Als 8. Schritt ist eine Prüfung sinnvoll, ob die Verbindung vom komplementären Ausgang des zuletzt gerouteten Schalters (hier Schalter 4.2) mit dem korrespondierenden Eingang des ersten Schalters übereinstimmt (hier Ausgang 5 auf Eingang 1, beide sind in der unteren Hälfte). Die **Abbildung 5.26** zeigt das Beneš-Netz mit den in der Tabelle 5.3 ermittelten Schalterstellungen aus der Permutation *p* in Gleichung 5.23. Nachdem nun die Schalterstellungen äußeren Stufen bestimmt sind, wird in der gleichen Weise sukzessiv mit den inneren Stufen weiterverfahren, bis alle Schalterstellungen im Netz bestimmt sind.

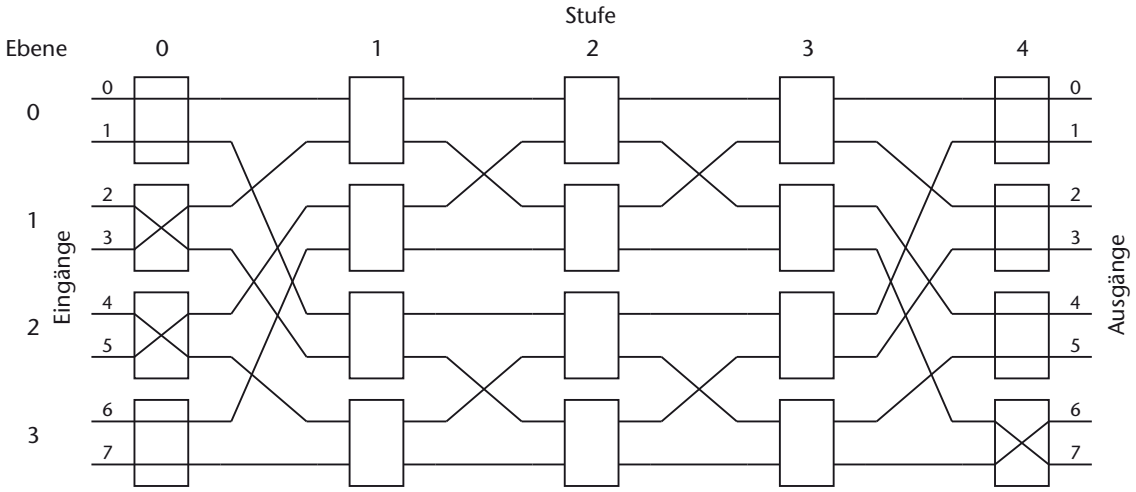


Abbildung 5.26: Ermittelte Schalterstellungen der beiden äußeren Stufen 0 und 4

Anzahl der Schritte im Looping-Routing. Die Anzahl der Schalter im Beneš-Netz ergibt sich aus der Anzahl der Schalterstufen (= $2\log_2 N$-1) plus eins, da die mittlere Stufe eine eigene Sequenz darstellt, und der Anzahl der Schalter pro Stufe (N/2). Da die Schalterstellungen im Looping-Routing sequentiell bestimmt werden, sind

$$m_{looping} = 2\log_2 N \cdot \frac{N}{2} = N \log_2 N \qquad (5.24)$$

Schritte nötig, um alle Schalterstellungen im Beneš-Netz zu bestimmen. Für ein Netzwerk mit $N = 8$ Prozessoren kann das Routing in überschaubaren 24 Schritten ausgeführt werden. Für ein Netzwerk mit einer Größe von $N = 512$ Prozessoren sind dagegen bereits 4608 (!) sequentielle Schritte notwendig (vgl. **Abbildung 5.27**). Davon ausgehend, dass im Handshake-Betrieb bei einem 32 Bit breiten Datenkanal für die Übertragung einer 32 Byte langen Nachricht insgesamt 16 Takte nötig sind, stehen demgegenüber 4608 Takte, um die Verbindung zu vermitteln. Das heißt, dass 99,7 % der Latenz bei der Übertragung einer 32 Byte großen Nachricht allein für das Routing der Verbindung benötigt werden.

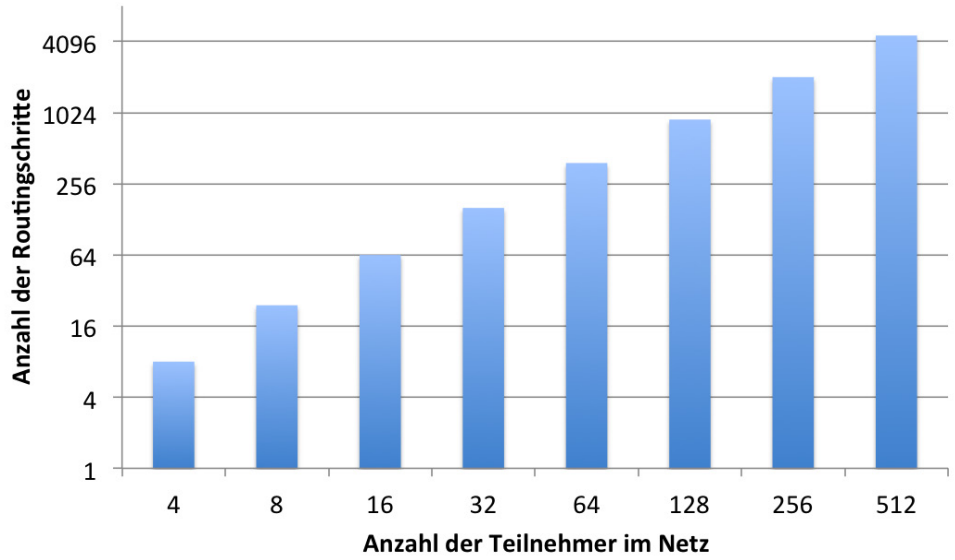


Abbildung 5.27: Anzahl der Routingschritte in Abhängigkeit von der Netzgröße

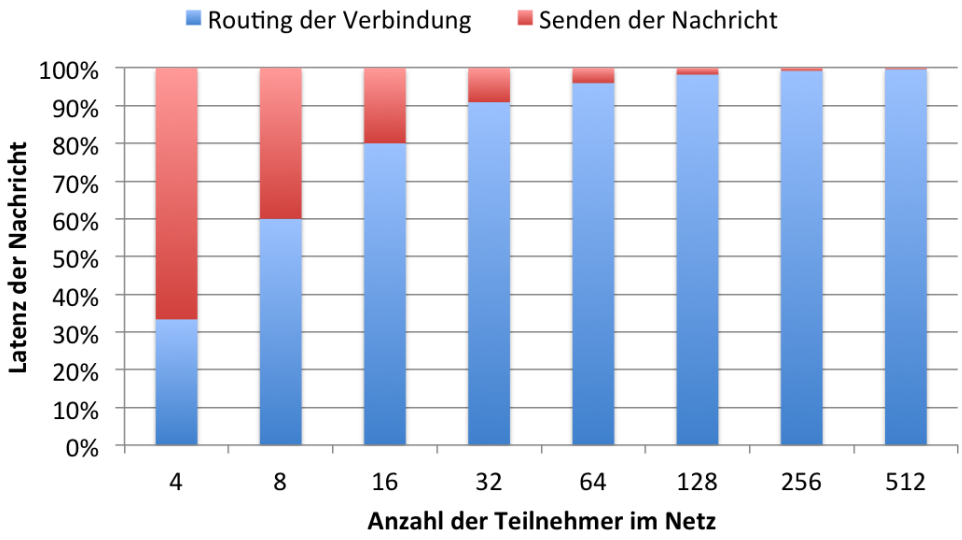


Abbildung 5.28: Anteil des Looping-Routings an der Latenz der Nachricht

Für Echtzeitanwendungen nicht geeignet. Die **Abbildung 5.28** zeigt den zeitlichen Anteil der Leitungsvermittlung und der eigentlichen Nachrichtenübertragung bei Netzen verschiedener Größe. Bereits bei 16 Teilnehmern im Netz werden 80 % der Zeit für den Verbindungsaufbau benötigt. Ein solches Missverhältnis ist in einem Echtzeitsystem natürlich nicht akzeptabel. Dazu kommt noch die Tatsache, dass aufgrund des potenziellen Umordnens bereits bestehender Verbindungen bei jedem neuen Verbindungswunsch immer das gesamte Netz neu geroutet werden muss, d.h. dass die gesamte Sequenz durchlaufen werden muss. Dadurch stellt sich das bekannte Looping-Routing als zeitlich äußerst ineffizient heraus und ist somit für Echtzeitanwendungen nicht geeignet.

5.5.3.1 Parallelität

Paralleles Routing. Die Latenz der Nachrichtenvermittlung kann deutlich reduziert werden, wenn das Looping-Routing in Teilen parallel ausgeführt wird. Da das Verbindungsnetzwerk als NoC auf einem FPGA implementiert werden soll, können parallelisierbare Anteile im Routing-Algorithmus zur Laufzeit simultan in FPGA-Hardware ausgeführt werden[24]. Gegeben seien j Teile des Routingverfahrens mit dem jeweiligen Anteil k, die n-fach parallel ausgeführt werden können. Der durch Parallelisierung mögliche Speedup ergibt sich mit:

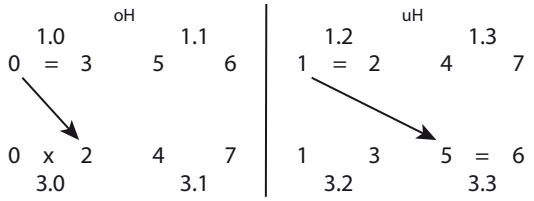
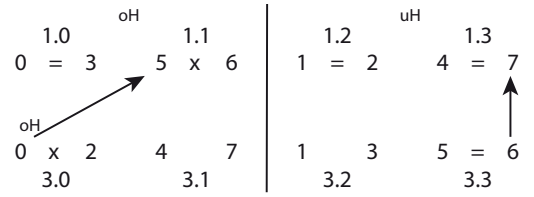
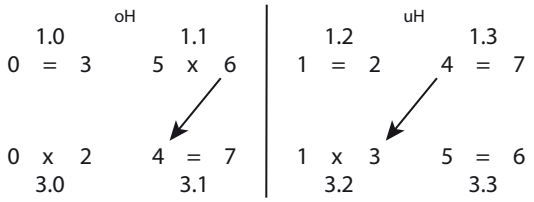
$$\frac{m_{sequentiell}}{m_{teilparallel}} = \frac{1}{1 - \sum_{i=1}^{j} k_i \left(1 - \frac{1}{n_i}\right)} \qquad (5.25)$$

Paralleles Routing der Teilnetze. Die Topologie des Beneš-Netzes besteht aus zwei zueinander spiegelbildlichen Baseline-Netzen. Eine entscheidende Besonderheit des Baseline-Netzes gegenüber anderen $\log_2 N$-Netzen wie dem Omega- oder dem Butterfly-Netz besteht in dessen rekursivem Aufbau aus kleineren Subnetzen [Ri97, S. 177]. Der Vorteil der Baseline-Topologie besteht nun also darin, dass sich die Vermaschung, also die Anzahl möglicher Zieladressen, mit jeder Stufe halbiert, bis in der letzten Stufe lediglich die komplementären Zieladressen der 2x2-Kreuzschalter übrig sind. Das bedeutet wiederum, dass die inneren Stufen im Beneš-Netz zunehmend parallel bearbeitet werden können, da dass Netz zur Mitte in immer kleinere Teilnetze zerfällt und auch keine weiteren Pfade zwischen

[24] gemäß dem Prinzip „divide et impera“

den Teilnetzen existieren. In dem in Abbildung 5.24 auf Seite 105 dargestellten Beispiel der Größe 8 x 8 zerfällt das Netz in den ersten inneren Stufen in zwei gleiche Hälften der Größe 4 x 4. Die **Tabelle 5.4** zeigt in Anknüpfung an Tabelle 5.3 auf Seite 108 f., wie die Teilnetze der oberen und unteren Hälfte (oH bzw. uH) teilweise parallel geroutet werden können. So kann im ersten Teilschritt zeitgleich mit dem Schalter 3.0 auch der Schalter 3.3 gesetzt werden, nachdem die Schalterstellungen der Schalter 1.0 und 1.2 willkürlich auf *gerade* (=) gesetzt wurden.

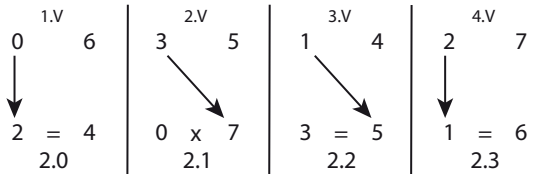
Tabelle 5.4: Paralleles Looping-Routing der ersten inneren Stufen im Beneš-Netz

Schritt 8: Die Schalter 1.0 (oH) und Schalter 1.2 (uH)der Eingänge 0 und 1 werden willkürlich auf „=“ gesetzt. Dadurch ergeben sich parallel für die Schalter 3.0 (oH) und 3.3 (uH) der korrespondierenden Ausgänge 2 und 5 die Stellung „=“.	
Schritt 9: Aus den Stellungen der Schalter 3.0 (oH) und 3.3 (uH) ergeben sich parallel die Stellungen der Schalter 1.1 (oH) und 1.3 (uH) mit den korrespondierenden Eingängen 5 und 7.	
Schritt 10: Aus den Stellungen der Schalter 1.1 (oH) und 1.3 (uH) ergeben sich parallel die letzten fehlenden Stellungen der Schalter 3.1 (oH) und 3.2 (uH) mit den korrespondierenden Ausgängen 4 und 3.	

Paralleles Routing der Mittelstufe. Im letzten Teilnetz, der Mittelstufe, können die Schalterstellungen der gesamten Stufe unabhängig von der Netzgröße vollständig parallel in einem einzigen Schritt bestimmt werden. Auch hier empfiehlt es sich, in einem zweiten abschließenden Schritt die komplementären Ein- bzw. Ausgänge auf ihre Richtigkeit zu prüfen. In dem dargestellten Beispiel der Größe 8 x 8 zerfällt

das Netz in der Mittelstufe in vier Viertel der Größe 2 x 2. Die **Tabelle 5.5** zeigt das parallele Routing der viertel Teilnetze (1.V, 2.V, 3.V und 4.V) in der Mittelstufe.

Tabelle 5.5: Paralleles Looping-Routing der Mittelstufe im Beneš-Netz

Schritt 11: Die Stellungen der Schalter 2.0 bis 2.3 ergeben sich aus je einem Eingang und dem dazu korrespondierenden Ausgang.	

Die **Abbildung 5.29** zeigt die ermittelten Schalterstellungen der Permutation aus Gleichung 5.23 auf Seite 107 für das gesamte Beneš-Netz.

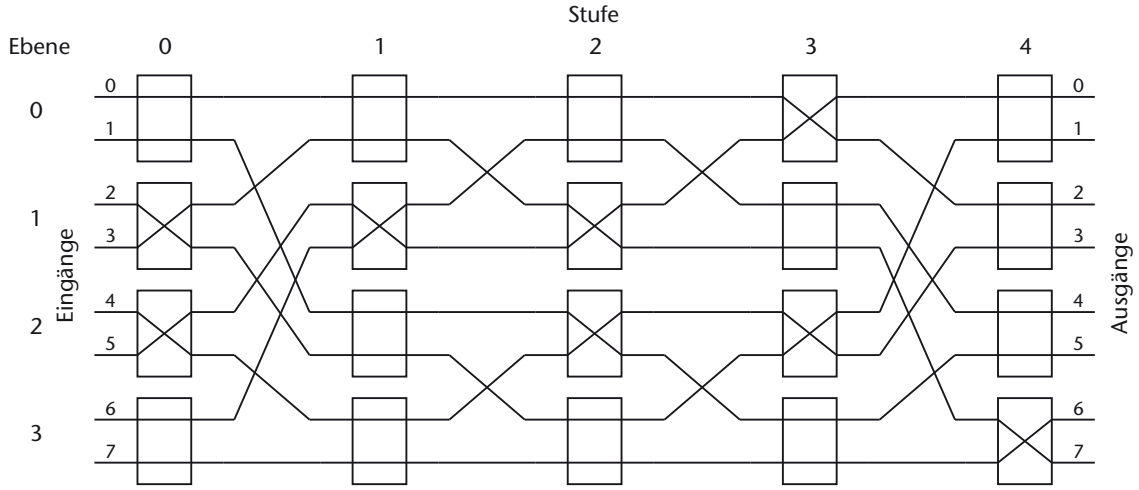


Abbildung 5.29: Ermittelte Schalterstellungen aller Stufen im Beneš-Netz

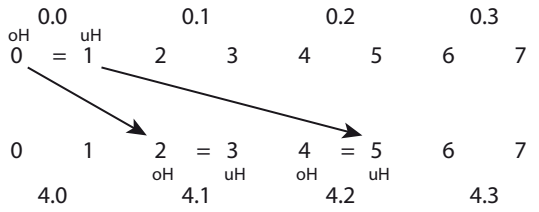
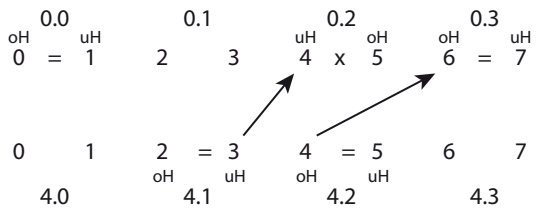
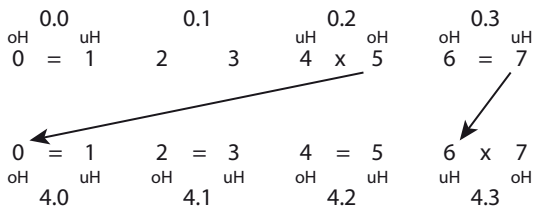
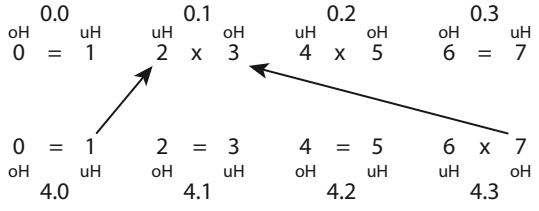
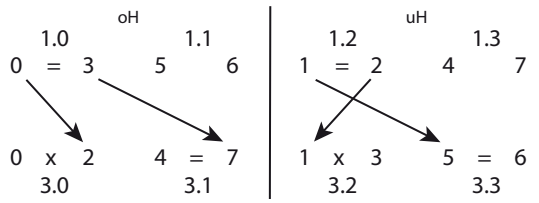
Während die Stellungen der Schalter in den beiden äußeren Stufen in 2*N* Schritten noch sequentiell bestimmt werden müssen, können die Schalterstellungen in den Subnetzen der nächstinneren Stufen bereits in 2*N*/2, 2*N*/4, usw. Schritten parallel bestimmt werden. Durch paralleles Routing der inneren Stufen (bzw. der Subnetze) reduziert sich die Anzahl *m* der Routing-Schritte im gesamten Beneš- Netz auf:

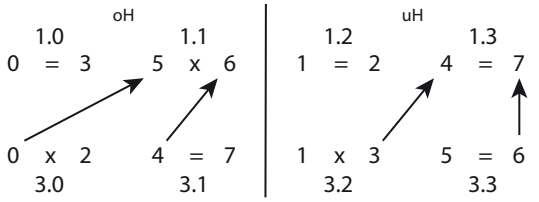
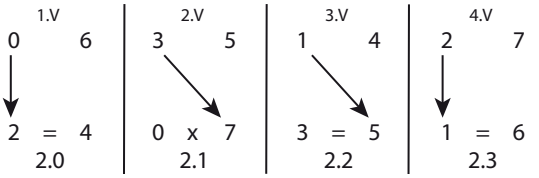
$$m_{looping} = \sum_{i=0}^{\log_2 N-1} \frac{N}{2^i} \qquad (5.26)$$

Für das oben eingeführte Beispiel der Übertragung einer 32 Byte langen Nachricht in einem Netzwerk mit 512 Prozessoren bedeutet das parallele Routing der inneren Subnetze eine Verkürzung der Routingdauer von 4608 auf 1022 Takte. Anteilig entfallen damit aber immer noch 98,5 % der Zeit auf das Routing der Verbindungen.

Paralleles Routing innerhalb der Teilnetze. Das Routing kann innerhalb der Teilnetze ebenfalls parallelisiert werden. Die **Tabelle 5.6** zeigt, wie das Routing aus Tabelle 5.3 bis Tabelle 5.5 auch innerhalb der Teilnetze parallel ablaufen kann.

Tabelle 5.6: Paralleles Looping-Routing im Beneš-Netz

Schritt 1: Der Schalter 0.0 wird willkürlich auf „=“ gesetzt. Dadurch ergibt sich parallel für die Schalter 4.1 und 4.2 der korrespondierenden Ausgänge 2 und 5 die Stellung „=“.	
Schritt 2: Aus den Stellungen der Schalter 4.1 und 4.2 ergeben sich parallel die Stellungen der Schalter 0.2 und 0.3 mit den korrespondierenden Eingängen 4 und 6.	
Schritt 3: Aus den Stellungen der Schalter 0.2 und 0.3 ergeben sich parallel die Stellungen der Schalter 4.0 und 4.3 mit den korrespondierenden Ausgängen 0 und 6.	
Schritt 4: Der Schalter 0.1 wird zeitgleich mit Hilfe der Stellungen der Schalter 4.0 und 4.3 gesetzt.	
Schritt 5: Die Schalter 1.0 (oH) und Schalter 1.2 (uH) werden willkürlich auf „=“ gesetzt. Dadurch ergeben sich parallel die Stellungen für die Schalter 3.0 und 3.1 (oH) sowie die Schalter 3.2 und 3.3 (uH).	

Schritt 6: Aus den Stellungen der Schalter 3.0 bzw. 3.1 (oH) und 3.2 bzw. 3.3 (uH) ergeben sich parallel die Stellungen der Schalter 1.1 (oH) und 1.3 (uH).	
Schritt 7: Die Stellungen der Schalter 2.0 bis 2.3 ergeben sich aus je einem Eingang und dem dazu korrespondierenden Ausgang.	

Durch teilparalleles Routing werden nun für das Routing des gesamten Beneš-Netzes genau so viele Schritte benötigt, wie im ursprünglich sequentiellen Routing nur für die beiden äußeren Stufen 0 und 4 nötig waren (vgl. Tabelle 5.3, S. 108). Die Anzahl m der notwendigen Schritte beträgt nun:

$$m_{looping} = \sum_{i=0}^{\log_2 N-1} \frac{N}{2 \cdot 2^i} \qquad (5.27)$$

Zusammenfassung. Die **Abbildung 5.30** zeigt die Latenz der Nachricht aufgrund der notwendigen Schritte im Looping-Routing für Netzwerke verschiedener Größe. Durch parallele Bearbeitung der Routing-Schritte konnte die Anzahl der notwendigen Schritte um bis zu 90 % gesenkt werden, wobei der durch Parallelität

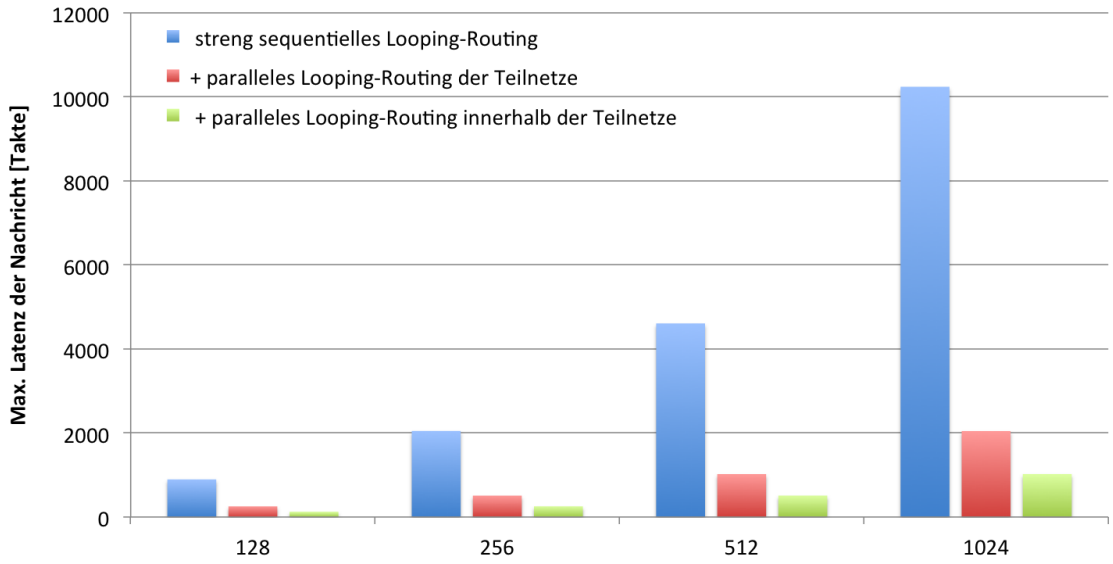


Abbildung 5.30: Max. Latenz der Nachricht durch Routing in Beneš-Netzen verschiedener Größe

erreichbare Speedup mit der Netzgröße um $O(\log_2 N)$ steigt. Die Latenz der Nachricht aus dem obigen Beispiel mit 512 Prozessoren wird durch teilparalleles Routing von ursprünglich 4608 auf nun 511 Takte verkürzt. Trotzdem werden in Relation immer noch 96,8 % der Übertragungszeit einer 32 Byte langen Nachricht für das Routing der Verbindung aufgewendet.

5.5.3.2 Pipelining

Warum Pipelining? Trotz der deutlichen Optimierungen durch die teilweise Parallelisierung bleibt das Looping-Routing ein Routingverfahren mit relativ vielen sequentiellen Einzelschritten, wodurch eine äußerst hohe Latenz bei der Übertragung einer einzelnen Nachricht entsteht, da für jeden auftretenden Verbindungswunsch das gesamte Netz neu geroutet werden muss. Ist gerade ein Routing für einen früheren Verbindungswunsch aktiv, so muss entweder das aktuelle Routing abgebrochen und neu gestartet werden oder die Latenz der Nachricht erhöht sich um die zusätzliche Wartezeit. Bei großen Netzen mit vielen Nachrichten ist so trotz Blockierungsfreiheit keine Kommunikation in Echtzeit möglich. Aus diesem Grund soll das Looping-Routing neben der Parallelität durch Pipelining optimiert werden.

Pipelining im Looping-Routing. Das Prinzip des Pipelinings ist in [Bö06, S. 39] am praktischen Beispiel sehr anschaulich erklärt. Der Prozess des Looping-Routings wird in kleinere Teilprozesse zerlegt, die dann zeitgleich als Fließband verarbeitet werden können. Da sich das Looping-Routing an den Schalterstufen orientiert, soll das Pipelining stufenweise erfolgen. Das Beneš-Netz besteht aus $2\log_2 N$-1 Stufen, wobei abgesehen von der Mittelstufe immer zwei Stufen wechselseitig parallel geroutet werden. Die Pipeline bekommt somit eine Länge von $\log_2 N$ Stufen.

Verkürzung der Latenz. Durch die steigende Parallelität innerhalb der Stufen, dauert die erste Pipeline-Stufe mit $2N$ Schritten am längsten, während in der letzten Pipeline-Stufe alle Schalter parallel in $N/2$ Schritten geroutet werden. Die maximale zusätzliche Latenz, die sich aufgrund eines bereits begonnenen Routings ergibt, beträgt somit $2N$ Takte. Für das Beispiel mit 512 Prozessoren verkürzt sich die maximale Latenz einer Nachricht durch stufenweises Pipelining in $\log_2 N$ Stufen von 1022 auf 767 Takte, was angesichts der eingangs erwähnten 16 Takte für die eigentliche Nachrichtenübertragung keinen nennenswerten Gewinn darstellt.

Zusammenfassung. Durch Pipelining der Schalterstufen kann die Latenz der Nachricht trotz bereits begonnenem Routing früherer Verbindungswünsche um bis zu 25 % verkürzt werden. Trotzdem stehen die Kosten für die Vermittlung der Verbindung und der eigentlichen Datenübertragung bei kurzen Nachrichten noch immer in einem extremen Missverhältnis zu Ungunsten der Latenz.

5.5.3.3 Fazit

Sehr hohe Latenz bei der Leitungsvermittlung. Das Prinzip des Looping-Routings besteht in dem paarweisen Setzen der Schalterstufen, angefangen mit den beiden äußeren Stufen. Dabei pendelt das Verfahren zwischen linker und rechter Seite im Netz hin und her, bis alle Schalterstellungen beider Stufen gesetzt sind. Dieses Vorgehen führt jedoch zu einer sehr langen Sequenz aus Einzelschritten und damit in größeren Netzen ($N \geq 128$) zu einer extrem hohen Latenz bei der Leitungsvermittlung. Das Looping-Routing ist aus diesem Grund trotz aller gezeigten Optimierungsversuche nicht für die Interprozessorkommunikation in Echtzeit geeignet.

5.5.4 Separation-Routing

Routing im doppelten Baseline-Netz. Eine Alternative zum Looping-Routing ist das *Separation-Routing*, welches in [Ri87, S. 21-44] als Routingverfahren für das Richter-Netz beschrieben ist. Dabei handelt es sich um ein modulares Koppelnetz, welches aus Gründen der einfacheren Fertigung aus zwei hintereinander geschalteten Baseline-Netzen aufgebaut wurde (**Abbildung 5.31**). Das Separation-Routing teilt das doppelte Baseline-Netz in zwei Teile, eine Eingangsseite (links, 1. Teil) und eine Ausgangsseite (rechts, 2. Teil), welche unterschiedlich

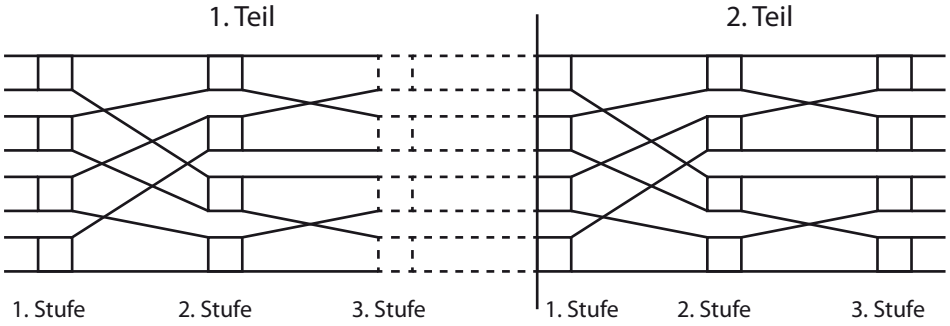


Abbildung 5.31: Modulares Koppelnetz[25]

[25] grafische Darstellung aus [Ri87, S. 24]

geroutet werden. Die Ausgangsseite besteht aus den rechten $\log_2 N$ Stufen des Koppelnetzes und beinhaltet somit ein nicht-blockierungsfreies Baseline-Netz, das wie ein $\log_2 N$-Netz, d.h. durch die stufenweise Auswertung der binären Zieladressen $I = [i_{n-1} \ldots i_0]_2$, geroutet wird[26]. Damit die rechte Seite (2. Teil) blockierungsfrei geroutet werden kann, müssen die Zieladressen der Eingänge in den ersten $\log_2 N$-1 Stufen (1. Teil) des Koppelnetzes entsprechend vorsortiert werden. Eine detaillierte Beschreibung zum Ablauf befindet sich in [Ri92].

Trennen komplementärer Zieladressen. Das in [Ri87] für das Richter-Netz beschriebene Verfahren beruht in der stufenweisen Trennung komplementärer Zieladressen in den ersten $\log_2 N$-1 Stufen des doppelten Baseline-Netzes. Das Ziel bei der Separation besteht darin, die komplementären Zieladressen[27] $I(s) = J(s) = [i_{n-1} \ldots i_{s+1}]_2$ in unterschiedliche Schalterausgänge zu routen. Einmal getrennt, können diese zwei Zieladressen aufgrund der Topologie des Baseline-Netzes nicht wieder zusammentreffen. Die kontrollierte Zusammenführung der Zieladressen erfolgt erst wieder in den rechten $\log_2 N$ Stufen (2. Teil). Der Beweis für die Blockierungsfreiheit im Separationsteil ist in [Ri87, S. 34-39] geführt worden.

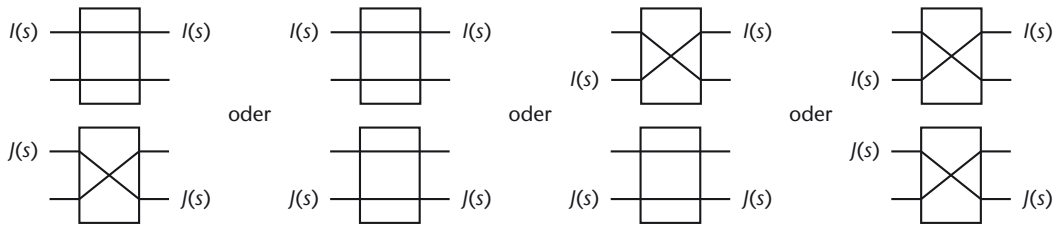


Abbildung 5.32: Trennen von komplementären Adressen [Ri87, S. 36]

Separation-Routing im Beneš-Netz. Neben der funktionalen und topologischen Äquivalenz, die für alle $\log_2$N-Netze gilt[28], besitzt das Baseline-Netz zudem die besondere Eigenschaft des identischen Routings der normalen und der inversen Variante [Ri97, S. 179 f.]. Aufgrund dieser besonderen Eigenschaft funktioniert das in [Ri87] und [Ri92] beschriebene Separation-Routing des modularen Koppel-Netzes in der gleichen Weise auch im Beneš-Netz, welches aus einem normalen und einem inversen Baseline-Netz aufgebaut ist, so dass das doppelte Baseline-Netz aus [Ri87] ohne Einschränkungen in ein Beneš-Netz überführt werden kann.

[26] mit $n = \log_2 N$ (N = Anzahl der Teilnehmer im Netz)

[27] mit $n = \log_2 N$ und Nummer der auswertenden Schalterstufe $s = 0 \ldots \log_2 N$-2

[28] Beweis der funktionalen Äquivalenz in [Wu80]

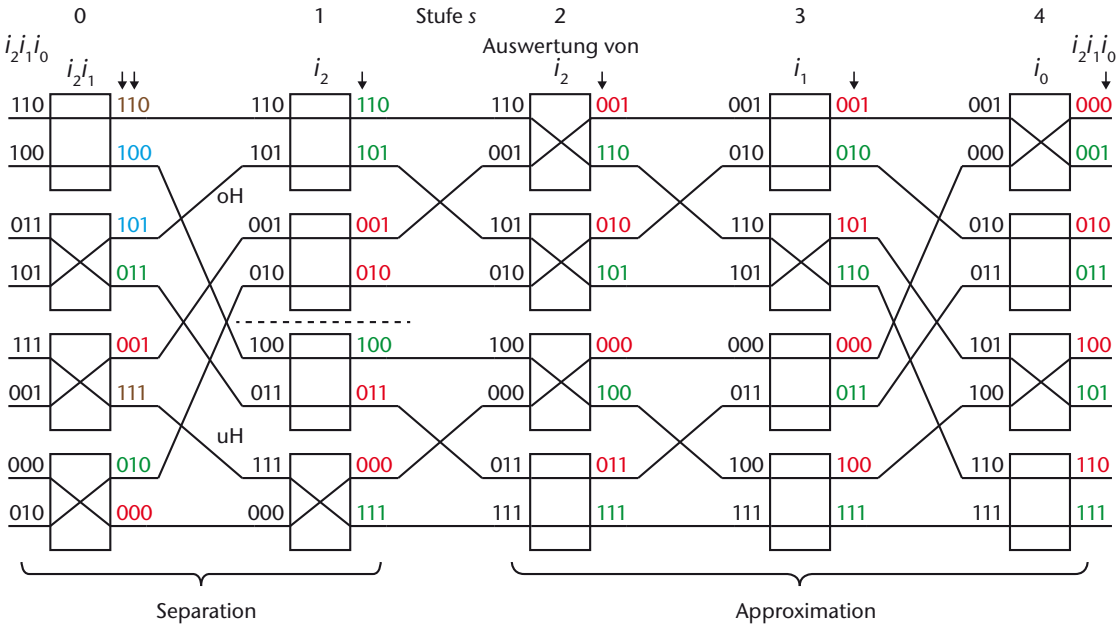


Abbildung 5.33: Separation-Routing im Beneš-Netz

Das in der **Abbildung 5.33** gezeigte Beneš-Netz besteht aus insgesamt fünf Schalterstufen, die mit $s = \{0\ldots4\}$ nummeriert sind. Links dargestellt sind die binären Zieladressen der Permutation aus Gleichung 5.28 in der Form $I = [i_2 i_1 i_0]_2$. Die Abbildung 5.33 zeigt die Funktion des Separation-Routings im Beneš-Netz der Größe 8 x 8 anhand der folgenden Beispiel-Permutation:

$$p = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 6 & 4 & 3 & 5 & 7 & 1 & 0 & 2 \end{pmatrix} \qquad (5.28)$$

Separationsteil. Der Sortieralgorithmus im Separationsteil funktioniert im Prinzip wie beim Looping-Routing. Beide Routing-Verfahren nutzen die rekursive Topologie des Beneš-Netzes, bei der sich die Vermaschung der inneren Stufen auf gleich große Teilnetze beschränkt und in den äußeren Stufen der obere Schalterausgang in das obere Teilnetz und der untere Schalterausgang in das untere Teilnetz verdrahtet ist[29]. Im Separationsteil besteht das Ziel darin, komplementäre Zieladressen an verschiedene Schalterausgänge (d.h. oberer bzw. unterer Ausgang) und damit in verschiedene Hälften (oH bzw. uH) der nachfolgenden Schalterstufe zu routen. Dies geschieht durch die stufenweise Separation von komplementären Zieladressen[30] $I(s) = J(s) = [i_{n-1} \ldots i_{s+1}]_2$ in den inneren Stufen. Aufgrund des rekursiven Aufbaus des Beneš-Netzes müssen die Schalter der ersten Stufe $s = 0$ in der

[29] vgl. Abbildung 5.24: Rekursiver Aufbau des Beneš-Netzes, S. 105

[30] mit $n = \log_2 N$ und Nummer der auswertenden Schalterstufe $s = 0\ldots\log_2 N\text{-}2$

Abbildung 5.33 so eingestellt werden, dass immer zwei komplementäre Adressen mit $I(s) = J(s) = [i_2 i_1]_2$ (gekennzeichnet durch gleiche Farbe) getrennt in die obere und die untere Hälfte der zweiten Stufe $s = 1$ geroutet werden. Dazu wird zunächst ein beliebiger Schalter der ersten Stufe, z.B. der Schalter oben links, auf die Stellung *gerade* gesetzt. Dieser Schalter besitzt im dargestellten Beispiel die beiden binären Zieladressen $I_x = [110]_2$ und $I_y = [100]_2$, wobei aufgrund der gewählten Schalterstellung I_x in die obere Hälfte und I_y in die untere Hälfte geroutet wird. Die beiden dazu komplementären Zieladressen $J_x = [111]_2$ und $J_y = [101]_2$ befinden sich am dritten bzw. am zweiten Schalter der ersten Schalterstufe und müssen nun nach unten bzw. nach oben geroutet werden. Daraus ergibt sich die Schalterstellung *gekreuzt* für den zweiten und den dritten Schalter der ersten Stufe. Die Stellung des vierten und letzten Schalter der ersten Stufe ergibt sich gleichzeitig aus den Zieladressen 011 des zweiten Schalters und 001 des dritten Schalters, welche zwangsläufig in die untere bzw. obere Hälfte geroutet werden. Die beiden Komplementäradressen 010 und 000 befinden sich beide am vierten Schalter der ersten Stufe und müssen in die obere Hälfte bzw. in die untere Hälfte geroutet werden, so dass sich für den vierten Schalter die Stellung *gekreuzt* ergibt. Damit ist die erste Schalterstufe geroutet. In der zweiten Schalterstufe setzt sich das Verfahren in gleicher Weise fort, allerdings wird hier ($s = 1$) nur noch das Bit i_2 der Zieladresse I ausgewertet. Sind alle Schalter der linken $\log_2 N$-1 Stufen gesetzt, so muss sich an den Eingängen der mittleren Schalterstufe für das höchstwertige Bit der Zieladresse I (hier Bit i_2) jeweils eine *Null* und eine *Eins* befinden. Für diesen Fall kann das $\log_2 N$-Netz der rechten Seite konfliktfrei geroutet werden[31].

Approximationsteil. Nachdem in den linken $\log_2 N$-1 Stufen des Netzes komplementäre Zieladressen getrennt wurden, werden diese nun in den rechten $\log_2 N$ Stufen des Netzes, dem Approximationsteil, sukzessive wieder zusammengeführt. Dies geschieht wie im normalen Baseline-Netz durch die bitweise Auswertung der binären Zieladressen, wobei eine *Null* immer an den oberen Schalterausgang und eine *Eins* immer an den unteren Schalterausgang geroutet wird. Das Ergebnis dieser Zusammenführung besteht darin, dass in der letzten Schalterstufe immer zwei komplementäre Zieladressen $I = J = [i_{n-1} \ldots i_1]_2$ an einem Schalter anliegen. In der Abbildung 5.33 sind die Zieladressen mit einer *Null* rot und die Zieladressen mit einer *Eins* grün markiert, so dass am Schalterausgang alle roten Adressen oben und alle grünen Adressen unten stehen.

[31] vgl. „Bedingung für blockierungsfreies Arbeiten“ in [Ri87, S. 28 ff.]

Fazit. Der große Vorteil des in [Ri87] beschriebenen Routingverfahrens besteht darin, dass die rechten $\log_2 N$ Stufen als selbstorganisierendes $\log_2 N$-Netz geroutet werden können. Aufgrund einer Anomalie des Baseline-Netzes kann das Verfahren vom Richter-Netz auch auf das Beneš-Netz angewendet werden. Dadurch beschränkt sich der sequentielle Anteil im Routing auf die linken $\log_2 N$-1 Stufen bzw. $N/2$ ($\log_2 N$-1) Schalter des Netzes. Zusammen mit der im Abschnitt 5.5.3 auf Seite 106 ff. beschriebenen teilweisen parallelen Abarbeitung voneinander unabhängiger Routingschritte sind im Separation-Routing insgesamt

$$m_{separation} = \sum_{i=1}^{\log_2 N-1} \frac{N}{2 \cdot 2^i} \qquad (5.29)$$

sequentielle Schritte nötig, um das blockierungsfreie Beneš-Netz zu routen. Damit halbiert sich im Vergleich zum Looping-Routing[32] die Anzahl der notwendigen sequentiellen Schritte, so dass in einem Netz mit $N = 512$ Prozessoren insgesamt „nur“ 256 Takte für das Routing benötigt werden. Wie die **Abbildung 5.34** zeigt, entfällt jedoch bei $N = 512$ Teilnehmern mit 93,7 % noch immer der weitaus größte Teil der Übertragungszeit einer 32 Byte großen Nachricht auf den Verbindungsaufbau, was für eine Übertragung von Echtzeitdaten bei weitem noch nicht effizient genug ist.

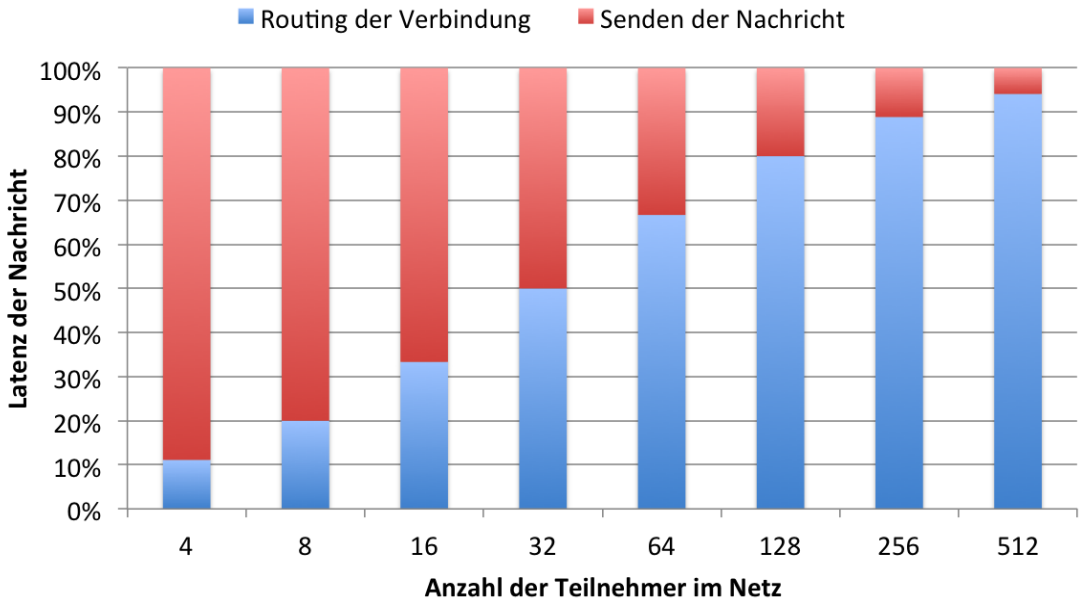


Abbildung 5.34: Anteil des Separation-Routings an der Latenz der Nachricht

[32] vgl. Abbildung 5.28: Anteil des Looping-Routings an der Latenz der Nachricht, S. 110

5.5.5 Routing im Schaltnetz

Inhalt. In diesem Abschnitt erfolgt mein Entwurf eines zeiteffizienten Routingverfahrens für das blockierungsfreie Beneš-Netz. Dieser Entwurf basiert auf den Erkenntnissen aus den vorangegangenen Abschnitten. Dort wurde gezeigt, dass blockierungsfreie Netze aufgrund alternativer Pfade in der Lage sind, zu jedem Zeitpunkt jeden Eingang mit jedem freien Ausgang zu verbinden. Allerdings ist bisher kein geeignetes Routingverfahren bekannt, um auch in großen Netzen mit $N \geq 128$ Teilnehmern alle möglichen Verbindungswünsche in so kurzer Zeit zu vermitteln, dass eine ausreichend schnelle Übertragung von zeitkritischen Echtzeitdaten möglich ist. Aus diesem Grund konzentriert sich der nun folgende Abschnitt auf das Routing von Verbindungswünschen mit maximaler Zeiteffizienz im blockierungsfreien Netz. Das Ergebnis dieser Arbeit wurde von mir selbst prototypisch auf verschiedenen Xilinx FPGAs implementiert und getestet[33]. Dieser Abschnitt beinhaltet im Einzelnen:

- Definition der Topologie und der Bezeichner im Netzwerk
- Erfassen aller auftretenden Fälle von Abhängigkeiten im Netzwerk
- Definition der Funktion eines Schaltnetzes, welches die Schalterstellungen im Netzwerk festlegt
- Test der Funktion des Schaltnetzes für ein Netzwerk der Größe $N = 16$ anhand von exemplarisch ausgewählten Permutationen
- Routing bei unvollständigen bzw. ungültigen Permutationen mit Hilfe eines zusätzlichen Bits zur Validierung der Zieladresse
- Angabe der Komplexität der Hardware
- Implementierung des Schaltnetzes in FPGA-Hardware

5.5.5.1 Aufbau

Topologie. Die topologische Basis für das Routingverfahren bildet das blockierungsfreie Beneš-Netz, welches durch Umordnen bestehender Verbindungen auf alternative Pfade in der Lage ist, zu jedem Zeitpunkt eine Verbindung von jedem Eingang zu jedem freien Ausgang herzustellen[34].

[33] vgl. Anhang A - Implementierung im FPGA, S. 249 ff.

[34] vgl. 5.5 Blockierungsfreie Netze, S. 100 ff.

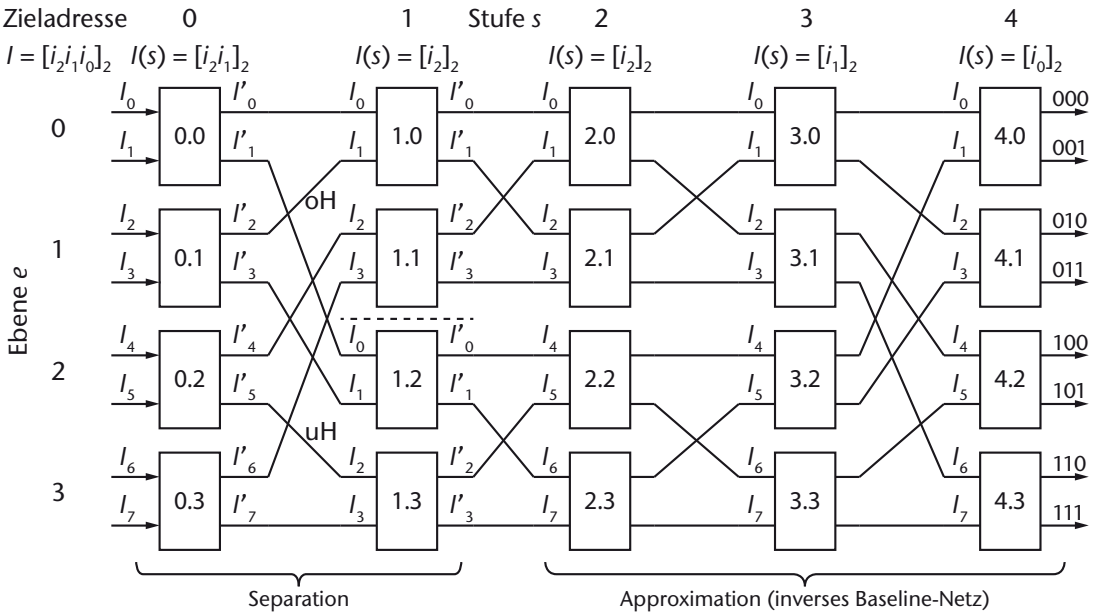


Abbildung 5.35: Bezeichnungen im Beneš-Netz der Größe 8 x 8

Bezeichnungen im Netz. Die **Abbildung 5.35** zeigt in einem Beispielnetz der Größe 8 x 8 die Bezeichnungen, wie sie in den nachfolgenden Ausführungen verwendet werden. Die Schalter werden im Netz mit *s.e* gekennzeichnet, wobei *s* die Nummer der Schalterstufe und *e* die Nummer der Schalterebene bezeichnet. Die Schalterstufen sind mit $s = \{0,1,\ldots,2(\log_2 N)-2\}$ fortlaufend von links nach rechts durchnummeriert. Die Schalterebenen sind mit $e = \{0,1,\ldots,(N/2)-1\}$ fortlaufend von oben nach unten durchnummeriert. Die Kennzeichnung der Zieladressen an den Eingängen des Netzes erfolgt mit I_j, wobei $j = \{0,1,\ldots,N-1\}$ fortlaufend von oben nach unten durchnummeriert wird. Jede Zieladresse besteht in binärer Form aus den Bits $I = [i_{n-1}\ldots i_0]_2$, wobei i_{n-1} das höchstwertige Bit (MSB[35]) und i_0 das niedrigstwertige Bit (LSB[36]) der Zieladresse bezeichnet. Da das Netzwerk in den linken $\log_2 N$-1 Stufen (Separationsteil) aufgrund der Unshuffle-Verdrahtung zu den inneren Stufen hin in zwei gleich große Teilnetze zerfällt[37] und das Routingverfahren die Abhängigkeit zwischen den Zieladressen innerhalb jedes Teilnetzes betrachtet, ist die Nomenklatur im linken Teil des Netzes so gewählt, dass sie sich in den unabhängigen Teilnetzen wiederholt (vgl. Abbildung 5.35 für $N = 8$ in der Stufe $s = 1$). In rechten $\log_2 N$ Stufen (Approximationsteil) besteht im Gegensatz zum linken Teil des Netzes keine Abhängigkeit zwischen den Schaltern, so dass hier die Eingänge jeder Stufe fortlaufend durchnummeriert sind.

[35] engl. *most significant bit*

[36] engl. *least significant bit*

[37] vgl. Abbildung 5.24: Rekursiver Aufbau des Beneš-Netzes, S. 105

5.5.5.2 Problemstellung

Hoher Zeitaufwand beim Routing. Der große Nachteil blockierungsfreier Netze gegenüber den nicht-blockierungsfreien Netzen besteht in dem recht aufwändigen Routing-Verfahren. Betrachtet man die Anzahl der notwendigen Schritte in den bisher bekannten Routingverfahren[38], so wird schnell klar, dass diese aufgrund ihres extrem ansteigenden Zeitbedarfs bei der Vermittlung von Verbindungen in größeren Netzen mit $N > 16$ Teilnehmern und der damit verbundenen Latenz der Nachricht nicht für die angestrebte Verarbeitung von Echtzeitdaten geeignet sind. Ein selbstorganisierendes Netzwerk mit einem dezentralen Routing auf Schalterebene wäre zwar schnell genug, kann aber bekanntermaßen aufgrund der Abhängigkeiten zwischen den Schaltern nicht alle auftretenden Verbindungswünsche abdecken[39] und ist somit hierfür nicht geeignet. Die Aufgabe besteht also darin, ein Routingverfahren zu finden, welches unabhängig von der Netzgröße einen möglichst geringen Zeitbedarf bei der Vermittlung von Nachrichten besitzt und gleichzeitig alle Verbindungswünsche von jedem Sender zu jedem Empfänger, d.h. alle möglichen $N!$ Permutationen der Eingänge auf die Ausgänge, vermitteln kann.

Ausgangspunkt. Das Problem des hohen Zeitbedarfs bei der Vermittlung von Nachrichten beruht ausschließlich auf dem hohen sequentiellen Anteil in den bisher betrachteten Routingverfahren. Als Ausgangspunkt für die nachfolgenden Überlegungen soll das *Separation-Routing*[40] aus dem Richter-Netz dienen, da in diesem Verfahren der sequentielle Anteil im Routing auf die linken $\log_2 N$-1 Stufen des Netzes beschränkt ist. Die rechten $\log_2 N$ Stufen des Netzes, also der überwiegende Teil der Schalter, können durch ein dezentrales Routing auf Schalterebene gesetzt werden.

Einsatz von FPGA-Hardware. Da das MPSoC bei Space-Sharing auf einem FPGA implementiert wird, macht es Sinn, die Vorteile der FPGA-Technologie zu nutzen. So erlauben FPGAs die Implementierung und Ausführung beliebiger logischer Verknüpfungen in Hardware. Gelingt es, ein *Schaltnetz* zu konstruieren, welches durch kombinatorische Logik alle Schalterstellungen im Netz bestimmen kann, so könnte das gesamte Routing innerhalb einer Gatterlaufzeit und damit weitaus schneller als eine Taktperiode erfolgen.

[38] vgl. 5.5 Blockierungsfreie Netze, S. 100 ff.

[39] vgl. 5.5.2 Dezentrales Routing, S. 102 ff.

[40] vgl. 5.5.4 Separation-Routing, S. 117 ff.

5.5.5.3 Separation im linken Teil des Netzes

Stufenweise Auswertung der Zieladressen. Beim *Separation-Routing*[41] erfolgt das Setzen der Kreuzschalter im Separationsteil über einen stufenweisen Vergleich der binären Zieladressen in der Form $I = [i_{n-1} \ldots i_0]_2$ mit $n = \log_2 N$ und N als Anzahl der Teilnehmer. Angefangen mit i_0 als LSB, nimmt die Anzahl der signifikanten Bits mit jeder Stufe um 1 ab, so dass die Zieladressen $I(s) = [i_{n-1} \ldots i_{s+1}]_2$ stufenabhängig mit $s = \{0, \ldots, \log_2 N - 2\}$ ausgewertet werden, bis in der letzten Stufe mit $s = \log_2 N$-2 einzig das Bit i_{n-1} als MSB übrig bleibt.

Trennen komplementärer Zieladressen. Das Prinzip des Separation-Routings besteht darin, zwei komplementäre Zieladressen[42] der Form $I(s) = J(s) = [i_{n-1} \ldots i_{s+1}]_2$ in einer Stufe von beliebigen Schaltereingängen an jeweils verschiedene Schalterausgänge, d.h. oberer und unterer Schalterausgang und vice versa, zu routen. Das Schaltnetz muss also in der Lage sein, alle Paare von komplementären Zieladressen an den Schaltereingängen zu finden und so an die Ausgänge der Schalter zu routen, dass immer eine Zieladresse an einen oberen Schalterausgang und deren komplementäre Zieladresse an einen unteren Schalterausgang geroutet wird. Die beiden komplementären Zieladressen $I(s)$ und $J(s)$ werden am Schalterausgang mit $I'(s)$ und $J'(s)$ bezeichnet.

Abhängigkeiten bei der Schalterstellung. Die Reihenfolge der zu bestimmenden Schalter hängt beim Routing im Richter-Netz wie auch beim Looping-Routing von der Permutation der Zieladressen an den Eingängen ab. Der nächste zu bestimmende Schalter ist bei beiden Verfahren immer derjenige, an dessen Eingang eine komplementäre Zieladresse des aktuell bestimmten Schalters anliegt. Dadurch werden die Abhängigkeiten zwischen den Schaltern automatisch berücksichtigt. Im Gegensatz dazu ist die *Reihenfolge der Schalter* bei einem Schaltnetz *fest vorgegeben*. Hierin besteht auch die größte Herausforderung bei der Konstruktion eines Schaltnetzes. Aus diesem Grund werden zunächst die möglichen Abhängigkeiten der Schalterstellungen im Netz untersucht, an denen später die korrekte Funktion des Schaltnetzes nachgewiesen werden soll. Dafür werden nachfolgend verschiedene Fälle konstruiert, die vom Schaltnetz erkannt und gelöst werden müssen. Dabei gilt, dass $I = I(s) \circ LSB(I)$ und $J = J(s) \circ LSB(J)$ und $I(s) = J(s)$ und $LSB(I) = \overline{LSB(J)}$.

[41] vgl. „3.5 Der linke Teil des Netzes“ in [Ri87, S. 30-39]

[42] mit $n = \log_2 N$ und Nummer der auswertenden Schalterstufe $s = 0 \ldots \log_2 N$-2

5.5.5.3.1 Fallunterscheidungen

Fall 1: Keine Abhängigkeiten. Im einfachsten Fall, der in **Abbildung 5.36** dargestellt ist, liegen die beide komplementären Adressen $I(s)$ und $J(s)$ am selben Schalter an. In diesem Fall kann die Schalterstellung frei gewählt werden, da in jedem Fall beide Adressen an verschiedene Ausgänge geroutet werden. Zudem bestehen keine weiteren Abhängigkeiten zu anderen Schaltern bzw. deren Zieladressen.

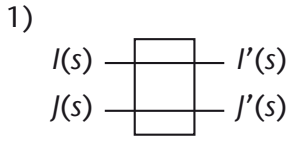


Abbildung 5.36: Keine Abhängigkeiten zu anderen Schaltern

Fall 2: Direkte Abhängigkeit zu einem vorhergehenden Schalter. Die Bestimmung der Schalterstellungen erfolgt im Separationsteil geordnet nach Stufen und Ebenen. Dabei kann der Fall eintreten, dass sich die zu $I(s)$ komplementäre Adresse $J(s)$ an einem Schalter befindet, dessen Stellung bereits bestimmt worden ist. In diesem Fall kann die Schalterstellung für $I(s)$ nicht frei gewählt werden. Entscheidend ist hier der Ausgang (oben oder unten) von $J'(s)$, da für $I'(s)$ der jeweils andere Ausgang gewählt werden muss. Dieser Fall ist in der **Abbildung 5.37** dargestellt, wobei für die Bezeichnung der Schalter gilt, dass *s.q* der zu bestimmende Schalter (fett dargestellt) und $p < q$ ist. Aufgrund der stufenweisen Auswertung muss dieser Fall für die Schalter der obersten Ebene $e = 0$ der Stufe nicht berücksichtigt werden. Das Gleiche gilt auch für die obersten Schalter jedes Teilnetzes, da zwischen den Teilnetzen keine Verdrahtungen und damit auch keine Abhängigkeiten existieren. Das heißt, dass bei einem Netz der Größe $N = 8$ (vgl. Abbildung 5.35) die Schalter 0.0, 1.0 und 1.2 willkürlich auf die Stellung *gerade* gesetzt werden.

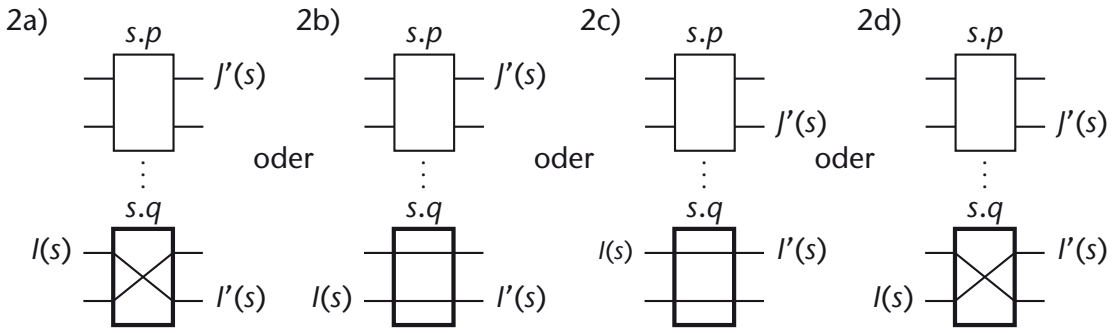


Abbildung 5.37: Direkte Abhängigkeit zu einem vorhergehenden Schalter

Fall 3: Indirekte Abhängigkeit zu einem vorhergehenden Schalter. Im Separation-Routing hängt die Reihenfolge der zu bestimmenden Schalter von den Zieladressen ab, d.h. es wird nachfolgend immer der Schalter mit der komplementären Adresse bestimmt. Im Schaltnetz dagegen ist die Reihenfolge der zu bestimmenden Schalter fest vorgegeben. Dabei können Abhängigkeiten auftreten, wie sie in der **Abbildung 5.38** dargestellt sind. Hier befindet sich eine Zieladresse $I'_x(s)$ am Ausgang eines bereits bestimmten Schalters $s.p$. Am zu bestimmenden Schalter $s.q$ mit $q > p$ befindet sich an einem Eingang die Zieladresse $I_y(s)$, die mit $I_x(s)$ in keinem direkten Zusammenhang steht, d.h. die beide Zieladressen sind nicht komplementär zueinander. An den Eingängen eines nachfolgenden Schalters $s.r$ dieser Stufe mit $r > q$

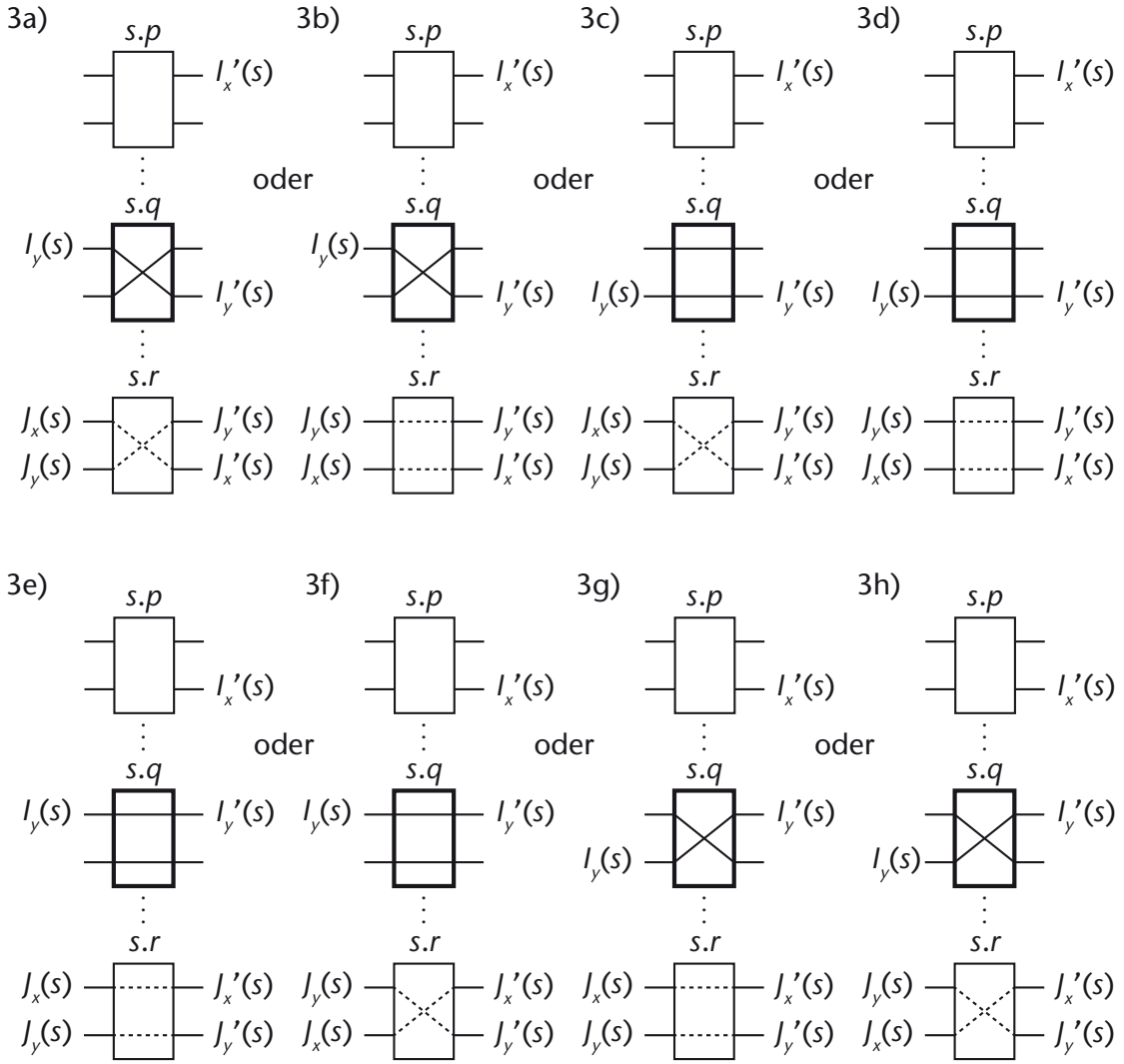


Abbildung 5.38: Indirekte Abhängigkeit zu einem vorhergehenden Schalter

befinden sich jedoch die beiden dazu komplementären Zieladressen $J_x(s)$ und $J_y(s)$. Da der Schalter *s.p* bereits bestimmt ist, steht implizit auch der Schalterausgang für $J'_x(s)$ und damit die Stellung des Schalters *s.q* fest. Aus der Position von $J'_x(s)$ ergibt sich wiederum die Position von $J'_y(s)$. Dies hat zur Folge, dass auch der Schalterausgang für $I'_y(s)$ und somit die Schalterstellung von *s.q* indirekt durch den Schalter *s.p* bestimmt wird. Zum Zeitpunkt der Auswertung von $I_y(s)$ ist der Schalter *s.r* jedoch noch nicht bestimmt.

Fall 4: Indirekte Abhängigkeit zu einem vorhergehenden und zu einem nachfolgendem Schalter. In dem in Fall 3 dargestellten Szenario befinden sich die beiden komplementären Adressen $J_x(s)$ und $J_y(s)$ am selben Schalter (*s.r)*. Daneben gibt es aber noch die Möglichkeit, dass diese beiden Adressen sich an verschiedenen Schaltern befinden, die aber indirekt voneinander abhängig sind. Diese Abhängigkeit ist in **Abbildung 5.40** bis **Abbildung 5.41** durch die beiden komplementären Adressen $I_z(s)$ und $J_z(s)$ hergestellt, welche sich an den nachfolgenden Schaltern *s.r* und *s.t* mit $t > r$ befinden. Während $I'_z(s)$ durch $J'_x(s)$ (an *s.r)* bzw. $I'_x(s)$ (an *s.p)* bestimmt ist, wird durch $J'_z(s)$ indirekt die Position von $J'_y(s)$ (an *s.t)* und damit auch von $I'_y(s)$ (an *s.q)* festgelegt.

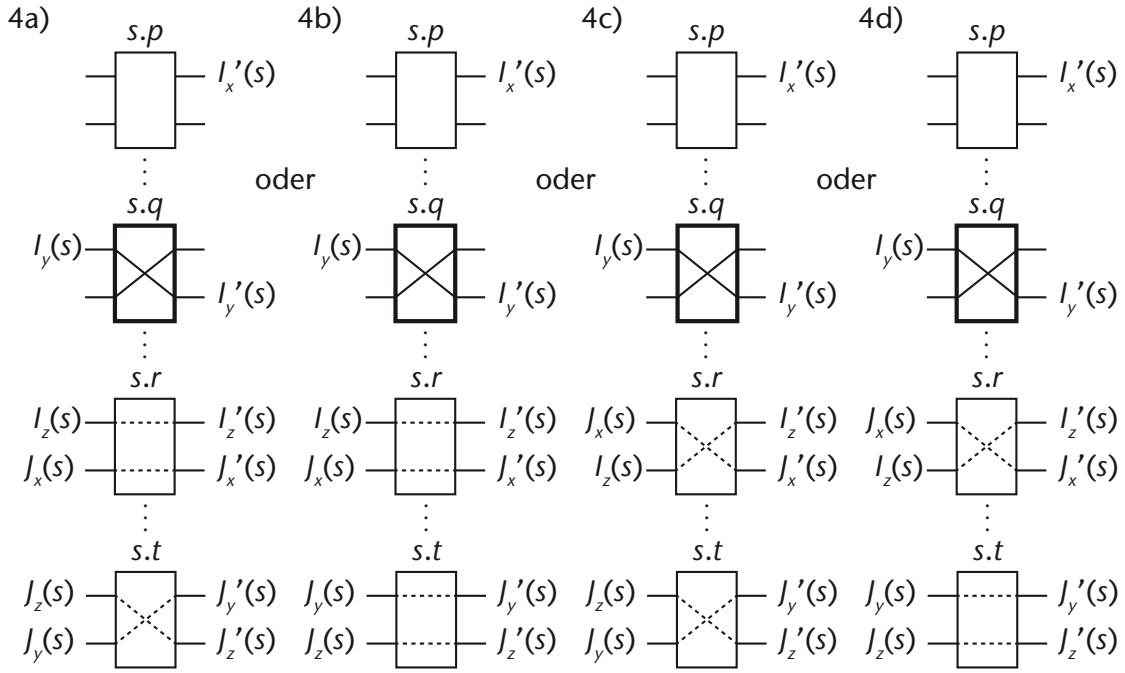


Abbildung 5.39: Indirekte Abhängigkeit zu einem vorhergehenden und zu einem nachfolgenden Schalter (1)

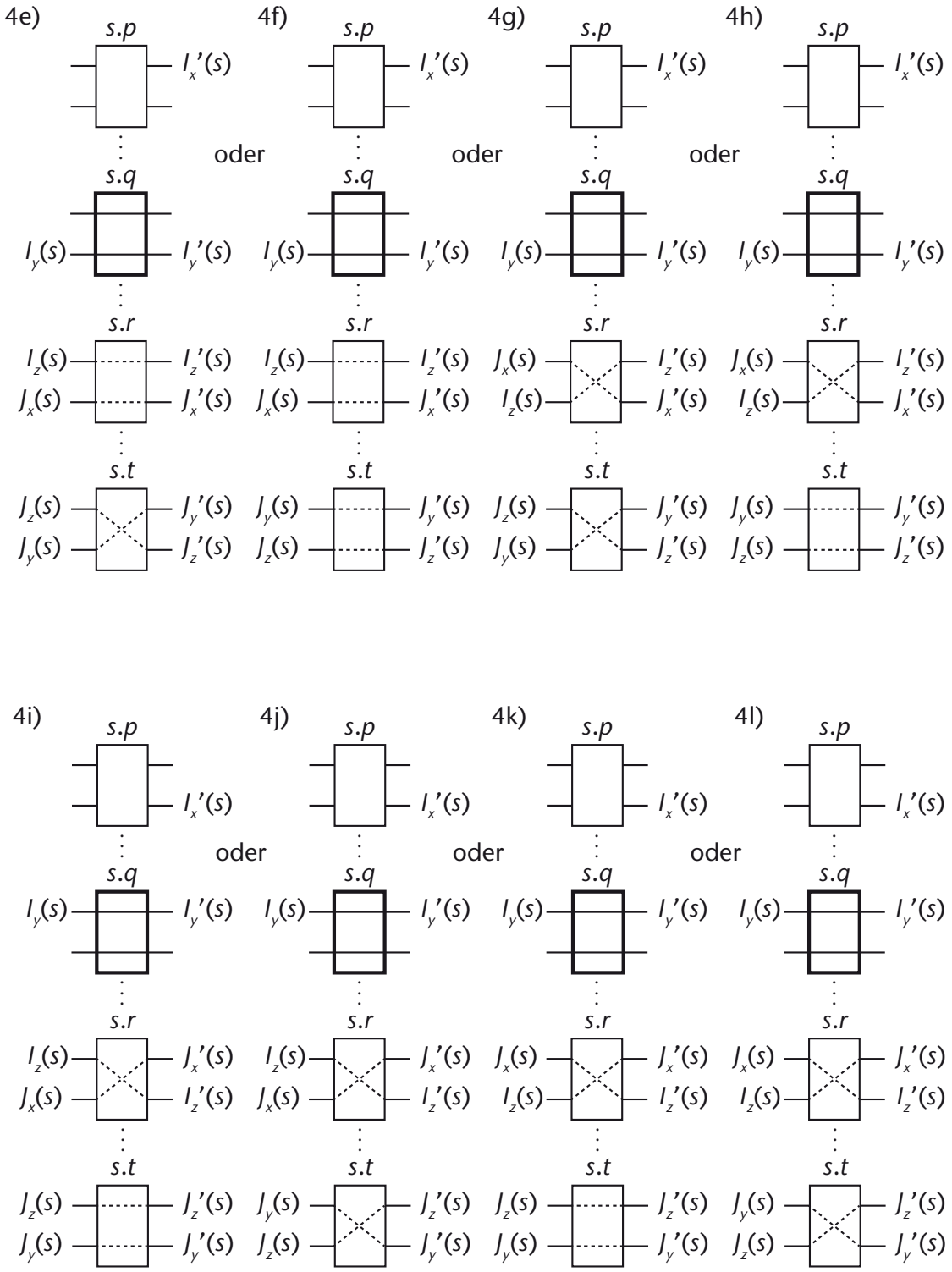


Abbildung 5.40: Indirekte Abhängigkeit zu einem vorhergehenden und zu einem nachfolgenden Schalter (2)

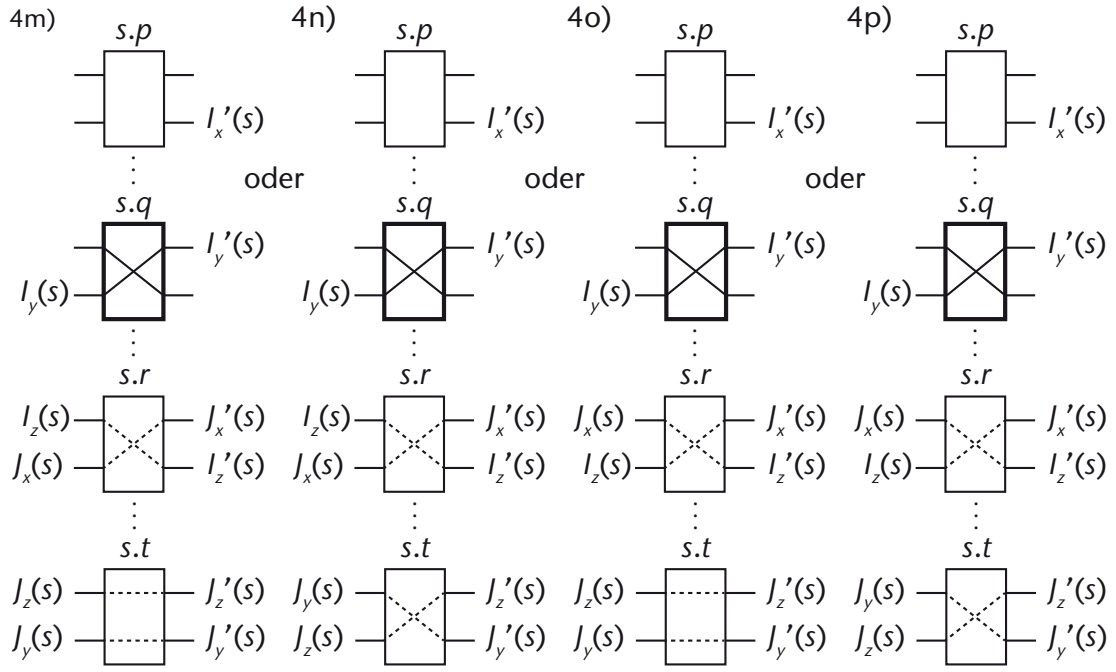


Abbildung 5.41: Indirekte Abhängigkeit zu einem vorhergehenden und zu einem nachfolgenden Schalter (3)

5.5.5.3.2 Finden von Abhängigkeiten und Auflösen von Konflikten

Finden von komplementären Zieladressen im Teilnetz. In jedem unabhängigen Teilnetz jeder Stufe existiert je nach Größe des Teilnetzes eine feste Anzahl von Zieladresspaaren mit denselben signifikanten Bits $I(s) = J(s) = [i_{n-1} \ldots i_{s+1}]_2$. In den linken Stufen, dem *Separationsteil*, besteht das Ziel darin, in jeder Stufe alle Paare mit komplementären Zieladressen zu finden und jedes Adressenpaar in verschiedene Schalterausgänge (oberer und unterer Ausgang) und damit in verschiedene Teilnetze der nachfolgenden Stufe zu routen[43]. Pro Teilnetz umfasst jeder Vergleich abhängig von der Stufe bzw. der Anzahl unabhängiger Teilnetze in der Stufe $N(s)$ Adressen[44], die Abhängigkeiten miteinander aufweisen, wobei pro Stufe $s + 1$ Teilnetze existieren, sodass in der Summe wieder N Adressen pro Stufe ausgewertet werden. Aufgrund der Unshuffle-Verdrahtung verdoppelt sich die Zahl unabhängiger Teilnetze mit jeder Stufe, während sich die Zahl an voneinander abhängigen Schalter halbiert und damit die Anzahl der signifikanten Bits in $I(s)$ um ein Bit verringert, sodass in der letzten Stufe mit $s = \log_2 N\text{-}2$ nur noch das MSB ausgewertet wird.

[43] vgl. Prinzip des Separation-Routings in 5.5.4 Separation-Routing, S. 117 ff.

[44] $N(s) = 2^{n-s}$ mit $s = \{0 \ldots \log_2 N\text{-}2\}$ als Nummer der auszuwertenden Stufe

Vorgehensweise. Die Aufgabe im Separationsteil besteht darin, alle Abhängigkeiten zwischen den Schaltern aufgrund von komplementären Zieladressen zu finden und mögliche Konflikte durch geeignete Schalterstellungen zu lösen. Dies erfolgt durch den stufenweisen Vergleich der signifikanten Bits $I(s)=[i_{n-1}\ldots i_{s+1}]_2$ der binären Zieladressen $I=[i_{n-1}\ldots i_0]_2$. Aufgrund der festen Reihenfolge der zu bestimmenden Schalter sind im Separationsteil für jede Stufe zwei gegenläufige Vergleiche nötig, wie die **Abbildung 5.42** zeigt. Während der erste Vergleich am obersten Schalter beginnt und sich nach unten fortsetzt, beginnt der zweite Vergleich am untersten Schalter und setzt sich vice versa nach oben fort. Im ersten Vergleich sind die signifikanten Bits der Zieladressen an den Schaltereingängen mit $I(s)$ und an den Schalterausgängen mit $I'(s)$ bezeichnet. Im zweiten Vergleich sind diese Bits an den Schaltereingängen mit $I'(s)$ und an den Schalterausgängen mit $I''(s)$ bezeichnet. Die temporären Schalterstellungen aus dem ersten Vergleich werden mit S' bezeichnet, während die endgültigen Schalterstellungen aus dem zweiten Vergleich mit S bezeichnet werden (vgl. Abbildung 5.42).

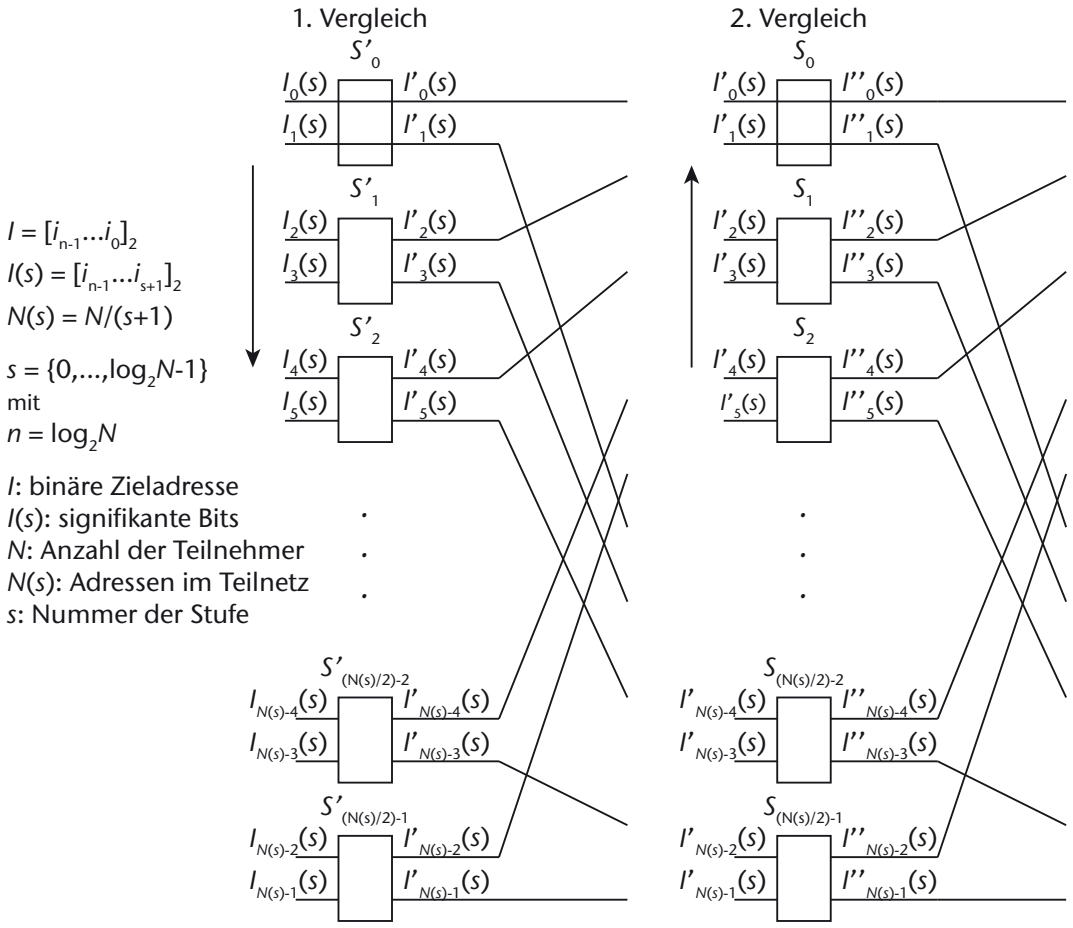


Abbildung 5.42: Finden von komplementären Zieladressen in einer Stufe/Teilnetz

5.5.5.3.3 Erster Vergleich

Komplementäre Zieladressen finden. Der erste Vergleich beginnt am obersten Schalter der Stufe bzw. des Teilnetzes (bei $s > 0$) und setzt sich dann nach unten fort (vgl. Abbildung 5.42). Wie die **Tabelle 5.7** zeigt, wird immer ein Eingang des Schalters mit den Ausgängen der darüberliegenden Schaltern der Stufe bzw. des Teilnetzes verglichen. Enthält der obere Eingang die komplementäre Adresse von einem der unteren Ausgänge, so muss der Schalter auf die Stellung *gerade* gesetzt werden ($g = 1$). Enthält der obere Eingang die komplementäre Adresse von einem der oberen Ausgänge, so muss der Schalter auf die Stellung *gekreuzt* gesetzt werden ($k = 1$). Entsprechend andersherum verhält es sich mit dem unteren Eingang, der ebenfalls mit den Ausgängen der darüberliegenden Schaltern verglichen wird.

Konflikte auflösen. Zunächst wird der oberste Schalter der Stufe bzw. des Teilnetzes willkürlich auf die Stellung $S'_0 = 0$ (*gerade)* gesetzt, so dass $I'_0(s) = I_0(s)$ und $I'_1(s) = I_1(s)$ ist. Weiter geht es mit dem zweiten Schalter S'_1 der Stufe bzw. des Teilnetzes, dessen Eingänge $I_2(s)$ und $I_3(s)$ mit den Ausgängen $I'_0(s)$ und $I'_1(s)$ des ersten Schalters auf die genannten Abhängigkeiten geprüft wird. Besteht eine Abhängigkeit, so wird der zweite Schalter S'_1 durch $g_{2/3}$ oder $k_{2/3}$ auf eine geeignete Stellung gesetzt und durch die Markierung $f_1 = 1$ fixiert (vgl. **Tabelle 5.8**). Besteht keine Abhängigkeit zum vorhergehenden Schalter, so wird die Stellung $S'_1 = 0$ (*gerade*) angenommen, in diesem Fall allerdings ohne dass eine Fixierung erfolgt ($f_1 = 0$). Die Schalterstellung von S_1 kann sich somit beim zweiten Vergleich noch ändern. Beim nun folgenden dritten Schalter S'_2 wird geprüft, ob bei dessen Eingängen $I_4(s)$ und $I_5(s)$ eine Abhängigkeit mit den Ausgängen $I'_0(s)$ bis $I'_3(s)$ des ersten bzw. des zweiten Schalters besteht. Es gilt jedoch immer die Einschränkung, dass der Vergleich nur mit Schaltern stattfindet, deren Stellung mit $f_e = 1$ fixiert ist. Besteht eine Abhängigkeit zwischen zwei der genannten Adressen, so wird auch der dritte Schalter entsprechend gesetzt und fixiert. Wie die Tabelle 5.7 zeigt, setzt sich dieses Vorgehen fort, bis alle $N(s)/2$ Schalter der Stufe bzw. des Teilnetzes geprüft und ggf. gesetzt sind. Im letzten Schritt werden demnach die Eingänge des letzten Schalters $S_{N(s)/2-1}$ der Stufe bzw. des Teilnetzes mit den Ausgängen aller anderen Schaltern der Stufe bzw. des Teilnetzes verglichen, soweit deren Stellung bereits fixiert ist. Die Ausgänge $I'(s)$ der Schalter ergeben sich aus deren Eingängen $I(s)$ und ihrer Stellung S' (vgl. Tabelle 5.8).

Tabelle 5.7: Finden von Abhängigkeiten (1. Vergleich)

***g* (Schalter *gerade*)**		***k* (Schalter *gekreuzt*)**	
g_0:	-	k_0:	-
g_1:	-	k_1:	-
g_2:	$I_2(s) == I'_1(s)$	k_2:	$I_2(s) == I'_0(s)$
g_3:	$I_3(s) == I'_0(s)$	k_3:	$I_3(s) == I'_1(s)$
g_4:	$I_4(s) == I'_1(s)$ or $(I_4(s) == I'_3(s))$ and f_1	k_4:	$I_4(s) == I'_0(s)$ or $(I_4(s) == I'_2(s))$ and f_1
g_5:	$I_5(s) == I'_0(s)$ or $(I_5(s) == I'_2(s))$ and f_1	k_5:	$I_5(s) == I'_1(s)$ or $(I_5(s) == I'_3(s))$ and f_1
…		…	
$g_{N(s)-4}$:	$I_{N(s)-4}(s) == I'_1(s)$ or $(I_{N(s)-4}(s) == I'_3(s))$ and f_1 or $(I_{N(s)-4}(s) == I'_5(s))$ and f_2 … or $(I_{N(s)-4}(s) == I'_{N(s)-5}(s))$ and $f_{N(s)/2-3}$	$k_{N(s)-4}$:	$I_{N(s)-4}(s) == I'_0(s)$ or $(I_{N(s)-4}(s) == I'_2(s))$ and f_1 or $(I_{N(s)-4}(s) == I'_4(s))$ and f_2 … or $(I_{N(s)-4}(s) == I'_{N(s)-6}(s))$ and $f_{N(s)/2-3}$
$g_{N(s)-3}$:	$I_{N(s)-3}(s) == I'_0(s)$ or $(I_{N(s)-3}(s) == I'_2(s))$ and f_1 or $(I_{N(s)-3}(s) == I'_4(s))$ and f_2 … or $(I_{N(s)-3}(s) == I'_{N(s)-6}(s))$ and $f_{N(s)/2-3}$	$k_{N(s)-3}$:	$I_{N(s)-3}(s) == I'_1(s)$ or $(I_{N(s)-3}(s) == I'_3(s))$ and f_1 or $(I_{N(s)-3}(s) == I'_5(s))$ and f_2 … or $(I_{N(s)-3}(s) == I'_{N(s)-5}(s))$ and $f_{N(s)/2-3}$
$g_{N(s)-2}$:	$I_{N(s)-2}(s) == I'_1(s)$ or $(I_{N(s)-2}(s) == I'_3(s))$ and f_1 or $(I_{N(s)-2}(s) == I'_5(s))$ and f_2 … or $(I_{N(s)-2}(s) == I'_{N(s)-3}(s))$ and $f_{N(s)/2-2}$	$k_{N(s)-2}$:	$I_{N(s)-2}(s) == I'_0(s)$ or $(I_{N(s)-2}(s) == I'_2(s))$ and f_1 or $(I_{N(s)-2}(s) == I'_4(s))$ and f_2 … or $(I_{N(s)-2}(s) == I'_{N(s)-4}(s))$ and $f_{N(s)/2-2}$
$g_{N(s)-1}$:	$I_{N(s)-1}(s) == I'_0(s)$ or $(I_{N(s)-1}(s) == I'_2(s))$ and f_1 or $(I_{N(s)-1}(s) == I'_4(s))$ and f_2 … or $(I_{N(s)-1}(s) == I'_{N(s)-4}(s))$ and $f_{N(s)/2-2}$	$k_{N(s)-1}$:	$I_{N(s)-1} == I'_1$ or $(I_{N(s)-1} == I'_3)$ and f_1 or $(I_{N(s)-1} == I'_5)$ and f_2 … or $(I_{N(s)-1} == I'_{N(s)-3})$ and $f_{N(s)/2-2}$

Tabelle 5.8: Auflösen von Konflikten (1. Vergleich)

f (fixiert)	Stellung S'	Ausgang I'(s)
f_0: 1	S'_0: 0	$I'_0(s)$: $I_0(s)$
		$I'_1(s)$: $I_1(s)$
f_1: g_2 or g_3 or k_2 or k_3	S'_1: k_2 or k_3	$I'_2(s)$: if S'_1 then $I_3(s)$ else $I_2(s)$
		$I'_3(s)$: if S'_1 then $I_2(s)$ else $I_3(s)$
f_2: g_4 or g_5 or k_4 or k_5	S'_2: k_4 or k_5	$I'_4(s)$: if S'_2 then $I_5(s)$ else $I_4(s)$
		$I'_5(s)$: if S'_2 then $I_4(s)$ else $I_5(s)$
…	…	…
$f_{N(s)/2-2}$: $g_{N(s)-4}$ or $g_{N(s)-3}$ or $k_{N(s)-4}$ or $k_{N(s)-3}$	$S'_{N(s)/2-2}$: $k_{N(s)-4}$ or $k_{N(s)-3}$	$I'_{N(s)-4}(s)$: if $S'_{N(s)/2-2}$ then $I_{N(s)-3}(s)$ else $I_{N(s)-4}(s)$
		$I'_{N(s)-3}(s)$: if $S'_{N(s)/2-2}$ then $I_{N(s)-4}(s)$ else $I_{N(s)-3}(s)$
$f_{N(s)/2-1}$: $g_{N(s)-2}$ or $g_{N(s)-1}$ or $k_{N(s)-2}$ or $k_{N(s)-1}$	$S'_{N(s)/2-1}$: $k_{N(s)-2}$ or $k_{N(s)-1}$	$I'_{N(s)-2}(s)$: if $S'_{N(s)/2-1}$ then $I_{N(s)-1}(s)$ else $I_{N(s)-2}(s)$
		$I'_{N(s)-1}(s)$: if $S'_{N(s)/2-1}$ then $I_{N(s)-2}(s)$ else $I_{N(s)-1}(s)$

5.5.5.3.4 Zweiter Vergleich

Komplementäre Zieladressen finden. Mit dem ersten Vergleich wurden Abhängigkeiten dadurch gefunden, dass die Eingänge jedes Schalters der Stufe bzw. des Teilnetzes auf komplementäre Zieladressen in den Ausgängen der darüberliegenden Schalter der Stufe bzw. des Teilnetzes geprüft wurde. Mit diesem Vergleich können jedoch nur die Fälle 2a bis 2d (vgl. Abbildung 5.37, S. 126) erkannt werden, bei denen eine direkte Abhängigkeit zu einem vorhergehenden Schalter besteht. Die Fälle 3 und 4, bei denen eine keine direkte Abhängigkeit zu einem vorhergehenden Schalter besteht, müssen nun in einem zweiten Vergleich ermittelt werden. Beim zweiten Vergleich ist die Reihenfolge der Schalter umgekehrt zur Schalterreihen-

folge des ersten Vergleichs, d.h. der zweite Vergleich beginnt am untersten Schalter der Stufe bzw. des Teilnetzes (bei $s > 0$) und setzt sich dann nach oben fort (vgl. Abbildung 5.42, S. 131). Wie die **Tabelle 5.9** zeigt, wird immer ein Eingang des Schalters mit den Ausgängen der darunterliegenden Schalter der Stufe bzw. des Teilnetzes verglichen. Enthält der obere Eingang die komplementäre Adresse von einem der oberen Ausgänge oder enthält der untere Eingang die komplementäre Adresse von einem der unteren Ausgänge, so muss der Schalter auf die Stellung *gekreuzt* gesetzt werden ($k' = 1$). Eine Prüfung der Stellung *gerade* wie auch die Fixierung der Schalter ist im zweiten Vergleich nicht notwendig.

Konflikte auflösen. Zunächst wird der unterste Schalter $S_{N(s)/2-1}$ der Stufe bzw. des Teilnetzes auf die Stellung $S'_{N(s)/2-1}$ gesetzt, so dass $I''_{N(s)-1}(s) = I'_{N(s)-1}(s)$ und $I''_{N(s)-2}(s) = I'_{N(s)-2}(s)$ ist. Weiter geht es mit dem zweituntersten Schalter $S_{N(s)/2-2}$ der Stufe bzw. des Teilnetzes, dessen Eingänge $I'_{N(s)-3}(s)$ bzw. $I'_{N(s)-4}(s)$ mit den Ausgängen $I''_{N(s)-1}(s)$ bzw. $I''_{N(s)-2}(s)$ des untersten Schalters auf die genannten Abhängigkeiten geprüft wird. Besteht eine Abhängigkeit, so wird der Schalter $S_{N(s)/2-2}$ durch $k_{N(s)-3}$ oder $k_{N(s)-4}$ auf die Stellung *gekreuzt* gesetzt (vgl. Tabelle 5.9). Besteht keine Abhängigkeit zum darunterliegenden Schalter, so wird mit $S_{N(s)/2-2} = S'_{N(s)/2-2}$ die Stellung aus dem ersten Vergleich übernommen. Wie die Tabelle 5.9 zeigt, setzt sich dieses Vorgehen fort, bis wieder alle $N(s)/2$ Schalter der Stufe bzw. des Teilnetzes geprüft und ggf. gesetzt sind. Im letzten Schritt werden demnach die Eingänge des ersten Schalters S_0 der Stufe bzw. des Teilnetzes mit den Ausgängen aller anderen Schaltern der Stufe bzw. des Teilnetzes verglichen. Die Ausgänge $I''(s)$ der Schalter ergeben sich aus deren Eingängen $I'(s)$ und ihrer Stellung S (vgl. Tabelle 5.9). Die Stellung *gerade* gilt automatisch, wenn die Stellung *gekreuzt* weder im ersten noch im zweiten Vergleich gesetzt worden ist.

5.5.5.3.5 Schalter setzen

Zieladressen zuweisen. Nachdem im zweiten Vergleich (Tabelle 5.9) die endgültigen Schalterstellungen ermittelt worden sind, können nun die kompletten Zieladressen I der Schaltereingänge an die Zieladressen I' der Schalterausgänge und damit zur Auswertung an die nachfolgende Schalterstufe (vgl. Abbildung 5.35, S. 123) zugewiesen werden. Diese Zuweisung ist in der **Tabelle 5.10** dargestellt.

Tabelle 5.9: Finden von Abgängigkeiten und Auflösen von Konflikten (2.Vergleich)

$k'_{I'(s)}$ (Schalter *gekreuzt*)	Stellung S	$I''(s)$
$k'_{N(s)-1}$: -	$S_{N(s)/2-1}$: $S'_{(N(s)/2)-1}$	$I''_{N(s)-1}(s)$: if $S_{N(s)/2-1}$ then $I_{N(s)-2}(s)$ else $I_{N(s)-1}(s)$
$k'_{N(s)-2}$: -		$I''_{N(s)-2}(s)$: if $S_{N(s)/2-1}$ then $I_{N(s)-1}(s)$ else $I_{N(s)-2}(s)$
$k'_{N(s)-3}$: $I'_{N(s)-4}(s) == I''_{N(s)-2}(s)$	$S_{N(s)/2-2}$: $S'_{(N(s)/2)-2}$ or $k'_{N(s)-4}$ or $k'_{N(s)-3}$	$I''_{N(s)-3}(s)$: if $S_{N(s)/2-2}$ then $I_{N(s)-4}(s)$ else $I_{N(s)-3}(s)$
$k'_{N(s)-4}$: $I'_{N(s)-3}(s) == I''_{N(s)-1}(s)$		$I''_{N(s)-4}(s)$: if $S_{N(s)/2-2}$ then $I_{N(s)-3}(s)$ else $I_{N(s)-4}(s)$
…	…	…
k'_5: $I'_5(s) == I''_7(s)$ … or $I'_5(s) == I''_{N(s)-3}(s)$ or $I'_5(s) == I''_{N(s)-1}(s)$	S_2: S'_2 or k'_4 or k'_5	$I''_5(s)$: if S_2 then $I_4(s)$ else $I_5(s)$
k'_4: $I'_4(s) == I''_6(s)$ … or $I'_4(s) == I''_{N(s)-4}(s)$ or $I'_4(s) == I''_{N(s)-2}(s)$		$I''_4(s)$: if S_2 then $I_5(s)$ else $I_4(s)$
k'_3: $I'_3(s) == I''_5(s)$ or $I'_3(s) == I''_7(s)$ … or $I'_3(s) == I''_{N(s)-3}(s)$ or $I'_3(s) == I''_{N(s)-1}(s)$	S_1: S'_1 or k'_2 or k'_3	$I''_3(s)$: if S_1 then $I_2(s)$ else $I_3(s)$
k'_2: $I'_2(s) == I''_4(s)$ or $I'_2(s) == I''_6(s)$ … or $I'_2(s) == I''_{N(s)-4}(s)$ or $I'_2(s) == I''_{N(s)-2}(s)$		$I''_2(s)$: if S_1 then $I_3(s)$ else $I_2(s)$
k'_1: -	S_0: 0	$I''_1(s)$: $I_1(s)$
k'_0: -		$I''_0(s)$: $I_0(s)$

Tabelle 5.10: Schalterausgänge setzen

I	S	I'	
I_0	S_0	I'_0:	I_0
I_1		I'_1:	I_1
I_2	S_1	I'_2:	if S_1 then I_3 else I_2
I_3		I'_3:	if S_1 then I_2 else I_3
I_4	S_2	I'_4:	if S_2 then I_5 else I_4
I_5		I'_5:	if S_2 then I_4 else I_5
…	…	…	
$I_{N(s)-4}$	$S_{N(s)/2-2}$	$I'_{N(s)-4}$:	if $S_{N(s)/2-2}$ then $I_{N(s)-3}$ else $I_{N(s)-4}$
$I_{N(s)-3}$		$I'_{N(s)-3}$:	if $S_{N(s)/2-2}$ then $I_{N(s)-4}$ else $I_{N(s)-3}$
$I_{N(s)-2}$	$S_{N(s)/2-1}$	$I'_{N(s)-2}$:	if $S_{N(s)/2-1}$ then $I_{N(s)-1}$ else $I_{N(s)-2}$
$I_{N(s)-1}$		$I'_{N(s)-1}$:	if $S_{N(s)/2-1}$ then $I_{N(s)-2}$ else $I_{N(s)-1}$

Verdrahtungsstufe. Nachdem im zweiten Vergleich alle Schalterstellungen der Stufe bestimmt und alle Zieladressen von den Eingängen an die Ausgänge der Schalter geroutet sind, folgt eine Verdrahtungsstufe, welche die Zieladressen der Ausgänge an die Eingänge der Schalter der nachfolgenden Schalterstufe routet. Aufgrund der Unshuffle-Verdrahtung erfolgt eine Aufteilung der Menge der Zieladressen in 2^s unabhängige Teilmengen, so dass in der nachfolgenden Stufe doppelt so viele Vergleiche, aber mit nur halb so vielen Zieladressen und einem signifikanten Bit weniger durchgeführt werden. Die **Abbildung 5.43** zeigt die Aufteilung der Menge der Zieladressen an die nachfolgende Schalterstufe, wobei durch die Verdrahtungsstufe zwischen den Schalterstufen die oberen Ausgänge der vorhergehenden Schalterstufe in die obere Hälfte der Eingänge der nachfolgenden Schalterstufe und die unteren Ausgänge der vorhergehenden Schalterstufe in die untere Hälfte der Eingänge der nachfolgenden Schalterstufe geroutet werden.

5.5.5.3.6 Funktionsnachweis

Nachweis der Funktion. Nachdem bisher die Funktion des Schaltnetzes am allgemeinen Fall erläutert wurde, soll nun der Nachweis der korrekten Funktion in den konkreten Fällen 2 bis 4 durchgeführt werden, wie sie in Abbildung 5.37

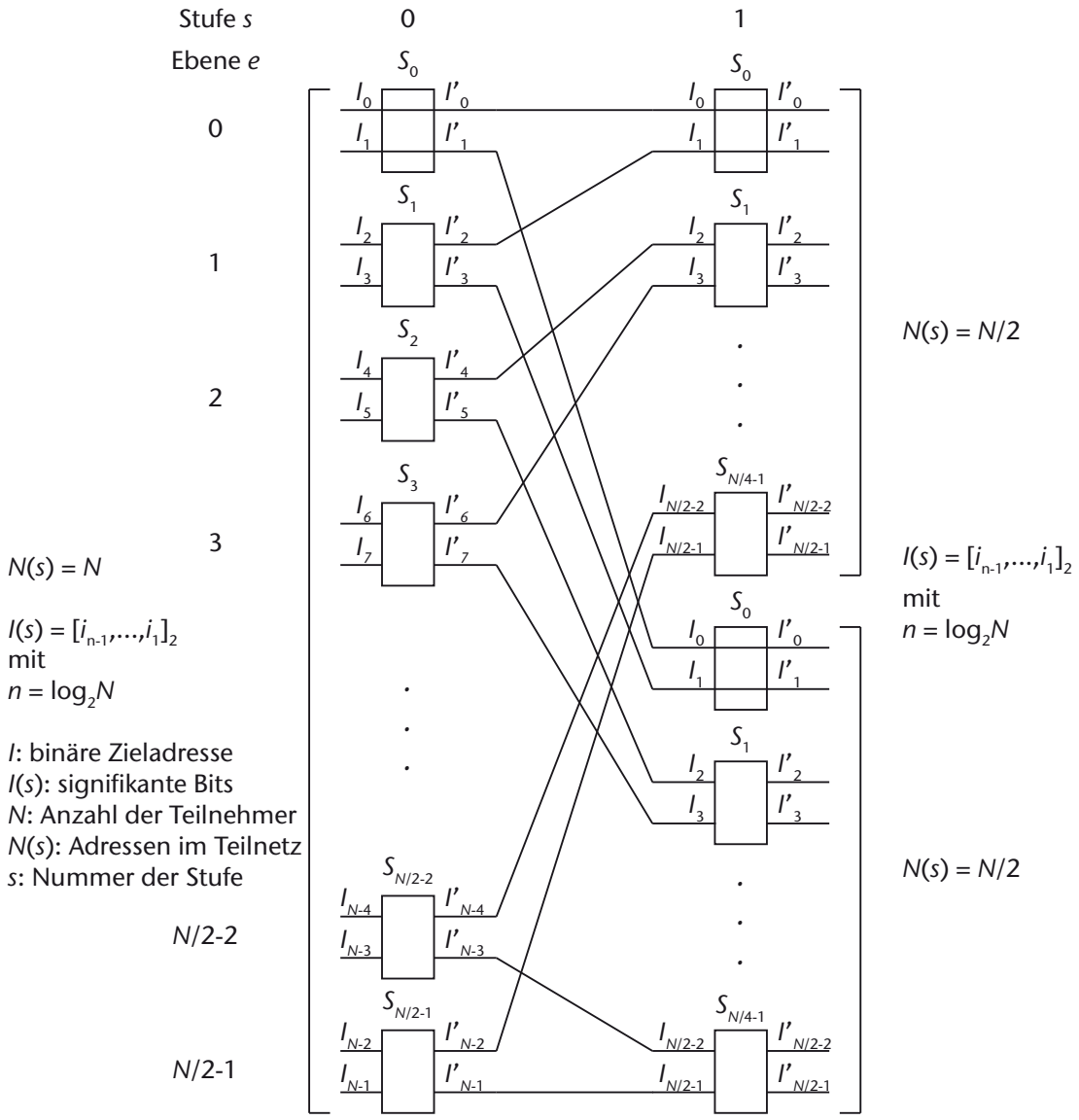


Abbildung 5.43: Stufenweise Aufteilung in unabhängige Teilnetze

bis Abbildung 5.41 dargestellt sind. Dieser Nachweis soll exemplarisch in einer Schalterstufe mit $N = 16$ Ein- und Ausgängen durchgeführt werden. Bei $N = 16$ Teilnehmern werden in der ersten Stufe des Netzes $(n-1)=3$ von 4 Bits der Zieladresse I in $I(s)$ ausgewertet. In **Tabelle 5.11** bis **Tabelle 5.23** sind die Ergebnisse der Fälle 2 bis 4 bei konkreten Beispielpermutationen mit allen Zwischenschritten dargestellt. Dabei soll im Detail gezeigt werden, dass das Schaltnetz in der Lage ist, die in den Fällen 2 bis 4 dargestellten Permutationen so zu routen, dass alle komplementären Adressen $I(s) = J(s)$ erkannt und an unterschiedliche Schalterausgänge und damit in unterschiedliche Hälften (oH bzw. uH) der nachfolgenden Schalterstufe geroutet werden.

Tabelle 5.11: Nachweis der Fälle 2a bis 2b

Fall	***I***		**1. Vergleich**					**2. Vergleich**				***I'***
			***I*(*s*)**	***g***	***k***	***f***	***S'***	***I'*(*s*)**	***k'***	***S***	***I'*'(*s*)**	
2a	J	3	001	-	-	1	0	001	-	0	001	3
		-	-	-	-			-	-		-	-
	I	2	001	0	1	1	1	-	0	1	-	-
		-	-	-	-			001	0		001	2
2b	J	6	011	0	0	0	0	011	0	0	011	6
		-	-	-	-			-	-		-	-
		-	-	-	-	0	0	-	-	0	-	-
	I	7	011	0	0			011	0		011	7
2c		-	-	-	-	0	0	-	-	0	-	-
	J	11	101	0	0			101	0		101	11
	I	10	101	0	0	0	0	101	0	0	101	10
		-	-	-	-			-	-		-	-
2d		-	-	-	-	0	1	-	-	1	111	14
	J	14	111	0	0			111	1		-	-
		-	-	-	-	0	0	-	-	0	-	-
	I	15	111	0	1			111	-		111	15

Tabelle 5.12: Nachweis der Fälle 3a und 3b

Fall	***I***		**1. Vergleich**					**2. Vergleich**				***I'***
			***I*(*s*)**	***g***	***k***	***f***	***S'***	***I'*(*s*)**	***k'***	***S***	***I'*'(*s*)**	
3a	I_x	0	000	-	-	1	0	000	-	0	000	0
		-	-	-	-			-	-		-	-
	I_y	2	001	0	0	0	0	001	1	1	-	-
		-	-	-	-			-	-		001	2
		-	-	-	-	-	-	-	-	-	-	-
		-	-	-	-			-	-		-	-
	J_x	1	000	0	1	1	1	001	0	1	001	3
	J_y	3	001	0	0			000	0		000	1
3b	I_x	8	100	0	0	0	0	100	0	0	100	8
		-	-	-	-			-	-		-	-
	I_y	10	101	0	0	0	0	101	1	1	-	-
		-	-	-	-			-	-		101	10
		-	-	-	-	-	-	-	-	-	-	-
		-	-	-	-			-	-		-	-
	J_y	11	101	0	0	0	0	101	-	0	101	11
	J_x	9	100	0	0			100	-		100	9

Tabelle 5.13: Nachweis der Fälle 3c und 3d

Fall	I		1. Vergleich					2. Vergleich				I'
			$I(s)$	g	k	f	S'	$I'(s)$	k'	S	$I''(s)$	
3c	I_x	0	000	-	-	1	0	000	-	0	000	0
		-	-	-	-			-	-		-	-
		-	-	-	-	0	0	-	-	0	-	-
	I_y	2	001	0	0			001	0		001	2
		-	-	-	-	-	-	-	-	-	-	-
		-	-	-	-			-	-		-	-
	J_x	1	000	0	1	1	1	001	0	1	001	3
	J_y	3	001	0	0			000	0		000	1
3d	I_x	8	100	0	0	0	0	100	0	0	100	8
		-	-	-	-			-	-		-	-
		-	-	-	-	0	0	-	-	0	-	-
	I_y	10	101	0	0			101	1		101	10
		-	-	-	-	-	-	-	-	-	-	-
		-	-	-	-			-	-		-	-
	J_y	11	101	0	0	0	0	101	-	0	101	11
	J_x	9	100	0	0			100	-		100	9

Tabelle 5.14: Nachweis der Fälle 3e und 3f

Fall	I		1. Vergleich					2. Vergleich				I'
			$I(s)$	g	k	f	S'	$I'(s)$	k'	S	$I''(s)$	
3e		-	-	-	-	1	0	-	-	0	-	-
	I_x	0	000	-	-			000	-		000	0
	I_y	2	001	0	0	0	0	001	1	0	001	2
		-	-	-	-			-	-		-	-
		-	-	-	-	-	-	-	-	-	-	-
		-	-	-	-			-	-		-	-
	J_x	1	000	1	0	1	0	000	0	0	000	1
	J_y	3	001	0	0			001	0		001	3
3f	I_x	8	100	0	0	0	0	100	0	0	100	8
		-	-	-	-			-	-		-	-
	I_y	10	101	0	0	0	0	101	1	1	-	-
		-	-	-	-			-	-		101	10
		-	-	-	-	-	-	-	-	-	-	-
		-	-	-	-			-	-		-	-
	J_y	11	101	0	0	0	0	101	-	0	101	11
	J_x	9	100	0	0			100	-		100	9

Tabelle 5.15: Nachweis der Fälle 3g und 3h

Fall	I		1. Vergleich					2. Vergleich				I'
			$I(s)$	g	k	f	S'	$I'(s)$	k'	S	$I''(s)$	
3g		-	-	-	-	1	0	-	-	0	-	-
	I_x	0	000	-	-			000	-		000	0
		-	-	-	-	0	0	001	1	1	001	2
	I_y	2	001	0	0			-	-		-	-
		-	-	-	-	-	-	-	-	-	-	-
		-	-	-	-			-	-		-	-
	J_x	1	000	1	0	1	0	000	0	0	000	1
	J_y	3	001	0	0			001	0		001	3
3h	I_x	8	100	0	0	0	0	100	0	0	100	8
		-	-	-	-			-	-		-	-
		-	-	-	-	0	0	-	-	0	-	-
	I_y	10	101	0	0			101	0		101	10
		-	-	-	-	-	-	-	-	-	-	-
		-	-	-	-			-	-		-	-
	J_y	11	101	0	0	0	0	101	-	0	101	11
	J_x	9	100	0	0			100	-		100	9

Tabelle 5.16: Nachweis der Fälle 4a und 4b

Fall	I		1. Vergleich					2. Vergleich				I'
			$I(s)$	g	k	f	S'	$I'(s)$	k'	S	$I''(s)$	
4a	I_x	0	000	-	-	1	0	000	-	0	000	0
		-	-	-	-			-	-		-	-
	I_y	2	001	0	0	0	0	001	1	1	-	-
		-	-	-	-			-	-		001	2
	I_z	4	010	0	0	1	0	010	0	0	010	4
	J_x	1	000	1	0			000	0		000	1
	J_z	5	010	0	1	1	1	001	0	1	001	3
	J_y	3	001	0	0			010	0		010	5
4b	I_x	8	100	0	0	0	0	100	0	0	100	8
		-	-	-	-			-	-		-	-
	I_y	10	101	0	0	0	0	101	1	1	-	-
		-	-	-	-			-	-		101	10
	I_z	12	110	0	0	0	0	110	0	0	110	12
	J_x	9	100	0	0			100	0		100	9
	J_y	11	101	0	0	0	0	101	-	0	101	11
	J_z	13	110	0	0			110	-		100	13

Tabelle 5.17: Nachweis der Fälle 4c und 4d

Fall	I		1. Vergleich					2. Vergleich				I'
			$I(s)$	g	k	f	S'	$I'(s)$	k'	S	$I''(s)$	
4c	I_x	0	000	-	-	1	0	000	-	0	000	0
		-	-	-	-			-	-		-	-
	I_y	2	001	0	0	0	0	001	1	1	-	-
		-	-	-	-			-	-		001	2
	J_x	1	000	0	0	1	1	010	0	1	010	4
	I_z	4	010	1	0			000	1		000	1
	J_z	5	010	0	1	1	1	001	0	1	001	3
	J_y	3	001	0	0			010	0		010	5
4d	I_x	8	100	0	0	0	0	100	0	0	100	8
		-	-	-	-			-	-		-	-
	I_y	10	101	0	0	0	0	101	1	1	-	-
		-	-	-	-			-	-		101	10
	J_x	9	100	0	0	0	0	100	0	1	110	12
	I_z	12	110	0	0			110	1		100	9
	J_y	11	101	0	0	0	0	101	-	0	101	11
	J_z	13	110	0	0			110	-		110	13

Tabelle 5.18: Nachweis der Fälle 4e und 4f

Fall	I		1. Vergleich					2. Vergleich				I'
			$I(s)$	g	k	f	S'	$I'(s)$	k'	S	$I''(s)$	
4e	I_x	0	000	-	-	1	0	000	-	0	000	0
		-	-	-	-			-	-		-	-
		-	-	-	-	0	0	-	-	0	-	-
	I_y	2	001	0	0			001	0		001	2
	I_z	4	010	0	0	1	0	010	0	0	010	4
	J_x	1	000	1	0			000	0		000	1
	J_z	5	010	0	1	1	1	001	0	1	001	3
	J_y	3	001	0	0			010	0		010	5
4f	I_x	8	100	0	0	0	0	100	0	0	100	8
		-	-	-	-			-	-		-	-
		-	-	-	-	0	0	-	-	0	-	-
	I_y	10	101	0	0			101	0		101	10
	I_z	12	110	0	0	0	0	110	0	0	110	12
	J_x	9	100	0	0			100	0		100	9
	J_y	11	101	0	0	0	0	101	-	0	101	11
	J_z	13	110	0	0			110	-		100	13

Tabelle 5.19: Nachweis der Fälle 4g und 4h

Fall	***I***		**1. Vergleich**					**2. Vergleich**				***I'***
			***I*(*s*)**	***g***	***k***	***f***	***S'***	***I'*(*s*)**	***k'***	***S***	***I"*(*s*)**	
4g	I_x	0	000	-	-	1	0	000	-	0	000	0
		-	-	-	-			-	-		-	-
		-	-	-	-	0	0	-	-	0	-	-
	I_y	2	001	0	0			001	0		001	2
	J_x	1	000	0	0	1	1	010	0	1	010	4
	I_z	4	010	1	0			000	0		000	1
	J_z	5	010	0	1	1	1	001	0	1	001	3
	J_y	3	001	0	0			010	0		010	5
4h	I_x	8	100	0	0	0	0	100	0	0	100	8
		-	-	-	-			-	-		-	-
		-	-	-	-	0	0	-	-	0	-	-
	I_y	10	101	0	0			101	0		101	10
	J_x	9	100	0	0	0	0	110	0	1	110	12
	I_z	12	110	0	0			100	1		100	9
	J_y	11	101	0	0	0	0	101	-	0	101	11
	J_z	13	110	0	0			110	-		100	13

Tabelle 5.20: Nachweis der Fälle 4i und 4j

Fall	***I***		**1. Vergleich**					**2. Vergleich**				***I'***
			***I*(*s*)**	***g***	***k***	***f***	***S'***	***I'*(*s*)**	***k'***	***S***	***I"*(*s*)**	
4i	-	-	-	-	-	1	0	-	-	0	000	0
	I_x	0	000	-	-			000	-		-	-
	I_y	2	001	0	0	0	0	001	1	0	001	2
		-	-	-	-			-	-		-	-
	I_z	4	010	0	0	1	1	000	0	1	000	1
	J_x	1	000	0	1			010	0		010	4
	J_z	5	010	1	0	1	0	010	0	0	010	5
	J_y	3	001	0	0			001	0		001	3
4j		-	-	-	-	0	0	-	-	1	100	8
	I_x	8	100	0	0			100	1		-	-
	I_y	10	101	0	0	0	0	101	1	1	-	-
		-	-	-	-			-	-		101	10
	I_z	12	110	0	0	0	0	110	0	0	110	12
	J_x	9	100	0	0			100	0		100	9
	J_y	11	101	0	0	0	0	101	-	0	101	11
	J_z	13	110	0	0			110	-		100	13

Tabelle 5.21: Nachweis der Fälle 4k und 4l

Fall	I		1. Vergleich					2. Vergleich				I'
			$I(s)$	g	k	f	S'	$I'(s)$	k'	S	$I''(s)$	
4k	-	-	-	-	-	1	0	-	-	0	-	-
	I_x	0	000	-	-			000	-		000	0
	I_y	2	001	0	0	0	0	001	0	0	001	2
		-	-	-	-			-	-			
	J_x	1	000	0	0	1	0	000	0	0	000	1
	I_z	4	010	1	0			010	0		010	4
	J_z	5	010	0	1	1	0	010	0	0	010	5
	J_y	3	001	0	0			001	0		001	3
4l		-	-	-	-	0	0	-	-	1	100	8
	I_x	8	100	0	0			100	1		-	-
	I_y	10	101	0	0	0	0	101	1	1	-	-
		-	-	-	-			-	-		101	10
	J_x	9	100	0	0	0	0	100	0	1	110	12
	I_z	12	110	0	0			110	1		100	9
	J_y	11	101	0	0	0	0	101	-	0	101	11
	J_z	13	110	0	0			110	-		110	13

Tabelle 5.22: Nachweis der Fälle 4m und 4n

Fall	I		1. Vergleich					2. Vergleich				I'
			$I(s)$	g	k	f	S'	$I'(s)$	k'	S	$I''(s)$	
4m	-	-	-	-	-	0	0	-	-	0	-	-
	I_x	0	000	-	-			000	0		000	0
		-	-	-	-	0	0	-	-	1	001	2
	I_y	2	001	0	0			001	1		-	-
	I_z	4	010	0	0	1	1	000	1	1	000	1
	J_x	1	000	0	1			010	0		010	4
	J_z	5	010	1	0	1	0	010	0	0	010	5
	J_y	3	001	0	0			001	0		001	3
4n		-	-	-	-	0	0	-	-	1	100	8
	I_x	8	100	0	0			100	1		-	-
		-	-	-	-	0	0	-	-	0	-	-
	I_y	10	101	0	0			101	0		101	10
	I_z	12	110	0	0	0	0	110	0	0	110	12
	J_x	9	100	0	0			100	0		100	9
	J_y	11	101	0	0	0	0	101	-	0	101	11
	J_z	13	110	0	0			110	-		100	13

Tabelle 5.23: Nachweis der Fälle 4o und 4p

Fall	I		1. Vergleich					2. Vergleich				I'
			$I(s)$	g	k	f	S'	$I'(s)$	k'	S	$I''(s)$	
4o	-	-	-	-	-	1	0	-	-	0	-	-
	I_x	0	000	-	-			000	-		000	0
		-	-	-	-	0	0	-	-	1	001	2
	I_y	2	001	0	0			001	1		-	-
	J_x	1	000	0	1	1	0	000	0	0	000	1
	I_z	4	010	0	0			010	0		010	4
	J_z	5	010	0	1	1	0	010	0	0	010	5
	J_y	3	001	0	0			001	0		001	3
4p		-	-	-	-	0	0	-	-	1	100	8
	I_x	8	100	0	0			100	1		-	-
		-	-	-	-	0	0	-	-	0	-	-
	I_y	10	101	0	0			101	0		101	10
	J_x	9	100	0	0	0	0	110	0	1	110	12
	I_z	12	110	0	0			100	1		100	9
	J_y	11	101	0	0	0	0	101	-	0	101	11
	J_z	13	110	0	0			110	-		100	13

5.5.5.4 Approximation im rechten Teil des Netzes

Zusammenführen der Zieladressen. Nachdem in den ersten $\log_2 N$-1 Stufen des Netzes, dem *Separationsteil*, komplementäre Zieladressen getrennt worden sind, erfolgt nun in den rechten $\log_2 N$ Stufen des Netzes, dem *Approximationsteil*[45], die stufenweise Zusammenführung der komplementären Zieladressen I und J mit $I = J = [i_{n-1} \ldots i_1]_2$ bis zum gemeinsamen Schalter in der letzten Stufe $s = 2\log_2 N$-1. Das Routing-Verfahren des Approximationsteils entspricht dem Routing eines Baseline-Netzes, d.h. angefangen mit dem MSB wird abhängig von der Stufe an jedem Schalter das Bit $i_{2\log_2 N-1-s}$ der binären Zieladresse $I = [i_{n-1} \ldots i_0]_2$ ausgewertet.

Auswertung auf Schalterebene. Im Vergleich zum Separationsteil bestehen im Approximationsteil keine Abhängigkeiten zwischen den Zieladressen verschiedener Schalter einer Stufe, so dass die Auswertung der Zieladressen dezentral bzw. in Form eines selbstorganisierenden Netzes auf Schalterebene implementiert werden kann und auch nur die Eingänge des zu bestimmenden Schalters betrachtet werden müssen.

[45] vgl. „3.4 Der rechte Teil des Netzes“ in [Ri87, S. 26-30]

Stufenweise Auswertung der binären Zieladressen. Wie die **Tabelle 5.24** zeigt, wird im Approximationsteil in jeder Stufe nur das Bit $i_{2\log_2 N-1-s}$ der binären Zieladresse ausgewertet. Aufgrund der vorhergehenden Sortierung der Zieladressen im Separationsteil können an den Eingängen jedes Schalters nur zwei verschiedene Situationen auftreten: entweder steht am oberen Eingang eine *Null* und gleichzeitig am unteren Eingang eine *Eins* oder anders herum. Steht am Eingang des Schalters in der Zieladresse eine *Null*, so muss dieser Eingang in den oberen Ausgang geroutet werden. Oder anders herum, steht in der Zieladresse eine *Eins*, so muss dieser Eingang in den unteren Ausgang geroutet werden. So wird der Schalter auf die Stellung *gerade* gesetzt, wenn am oberen Eingang eine *Null* steht und auf die Stellung *gekreuzt* gesetzt, wenn am oberen Eingang eine *Eins* steht. Es sei an dieser Stelle noch erwähnt, dass aufgrund der vorhergehenden Sortierung der Zieladressen für jeden Schalter nur ein Eingang ausgewertet werden müsste. Falls jedoch nur einzelne Verbindungswünsche bestehen[46], d.h. es existiert keine vollständige und gültige Permutation, so müssen immer beide Eingänge ausgewertet werden.

Tabelle 5.24: Routing im Approximationsteil

I	$I(s)$ mit $m = 2\log_2 N\text{-}1\text{-}s$	Stellung S	I'
I_0	$I_0(s)$: $I_0[i_m]_2$	S_0: $I_0(s) == 1$ or $I_1(s) == 0$	I'_0: if S_0 then I_1 else I_0
I_1	$I_1(s)$: $I_1[i_m]_2$		I'_1: if S_0 then I_0 else I_1
I_2	$I_2(s)$: $I_2[i_m]_2$	S_1: $I_2(s) == 1$ or $I_3(s) == 0$	I'_2: if S_1 then I_3 else I_2
I_3	$I_3(s)$: $I_3[i_m]_2$		I'_3: if S_1 then I_2 else I_3
I_4	$I_4(s)$: $I_4[i_m]_2$	S_2: $I_4(s) == 1$ or $I_5(s) == 0$	I'_4: if S_2 then I_5 else I_4
I_5	$I_5(s)$: $I_5[i_m]_2$		I'_5: if S_2 then I_4 else I_5
…	…	…	…
I_{N-4}	$I_{N-4}(s)$: $I_{N-4}[i_m]_2$	$S_{N/2-2}$: $I_{N-4}(s) == 1$ or $I_{N-3}(s) == 0$	I'_{N-4}: if $S_{N/2-2}$ then I_{N-3} else I_{N-4}
I_{N-3}	$I_{N-3}(s)$: $I_{N-3}[i_m]_2$		I'_{N-3}: if $S_{N/2-2}$ then I_{N-4} else I_{N-3}
I_{N-2}	$I_{N-2}(s)$: $I_{N-2}[i_m]_2$	$S_{N/2-1}$: $I_{N-2}(s) == 1$ or $I_{N-1}(s) == 0$	I'_{N-2}: if $S_{N/2-1}$ then I_{N-1} else I_{N-2}
I_{N-1}	$I_{N-1}(s)$: $I_{N-1}[i_m]_2$		I'_{N-1}: if $S_{N/2-1}$ then I_{N-2} else I_{N-1}

[46] vgl. 5.5.5.5 Routing unvollständiger Permutationen, S. 147 ff.

5.5.5.5 Routing unvollständiger Permutationen

Problem bei unvollständigen Abbildungen. In der praktischen Anwendung werden i.d.R. nicht alle Prozessoren zur gleichen Zeit miteinander kommunizieren. Das bedeutet, dass nicht alle Positionen der Zieladressen an den Eingängen der Schalter besetzt sind. Darüberhinaus könnte auch der Fall eintreten, dass zwei oder mehr Prozessoren dieselbe Zieladresse verwenden, da sie zeitgleich an denselben Prozessor senden möchten. Das bisher betrachtete Routing-Verfahren im Schaltnetz basiert auf dem Separation-Routing und funktioniert nur bei einer vollständigen Abbildung der Eingänge auf die Ausgänge, d.h. wenn die Zieladressen vollständig besetzt sind bzw. wenn keine Adresse fehlt oder mehrfach vorkommt.

Beispielpermutation. Die Gleichung 5.30 zeigt eine „Beispielpermutation", bei der nur die Zieladressen der Prozessoren 0, 2, 4, 5 und 6 besetzt sind. Die Prozessoren 1, 3 und 7 besitzen keinen Kommunikationswunsch, weshalb die Permutation an diesen Positionen undefiniert ist. Zudem besitzen die Prozessoren 4 und 6 die gleiche Zieladresse, so dass dieser Verbindungswunsch nicht zu routen wäre.

$$p = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & - & 6 & - & 5 & 3 & 5 & - \end{pmatrix} \tag{5.30}$$

Tauschen der Zieladresse. Damit das Routing im Schaltnetz trotzdem funktioniert, kann durch stetes Tauschen zweier Zieladressen immer eine gültige Permutation aufgebaut werden. Dabei wird nacheinander die Zieladresse, welche auf der Position des Senders steht, mit der Zieladresse des Empfängers getauscht, welche sich auf der Position eines beliebigen Sender befinden kann. Vorraussetzung für das Tauschen ist lediglich, dass die gewünschte Zieladresse nicht durch einen anderen Sender belegt ist. Im Falle derselben Zieladresse wie im Beispiel bei den Prozessoren 4 und 6 muss einer der beiden Verbindungswünsche solange zurückgestellt werden, bis die Zieladresse nicht mehr belegt ist. **Tabelle 5.25** zeigt, wie eine gültige Permutation durch das Tauschen der Zieladressen aufgebaut werden kann. Das Ergebnis entspricht dann dem Verbindungswunsch aus Gleichung 5.30. Im Fall der mehrfach auftretenden Zieladresse wird hier zunächst der Verbindungswunsch von Prozessor 4 erfüllt. Nach Beendigung der Datenübertragung durch Prozessor 4 kann auch die Verbindung von Prozessor 6 durch Tauschen der Zieladressen von Prozessor 4 und 6 aufgebaut werden.

Tabelle 5.25: Permutation durch Tauschen von Zieladressen

Verbindungs-wunsch	Tauschen der Zieladresse	Permutation
-	-	$p=\begin{pmatrix} 0&1&2&3&4&5&6&7 \\ 0&1&2&3&4&5&6&7 \end{pmatrix}$
P2 → P6	Pos. 2 ↔ Pos. 6	$p=\begin{pmatrix} 0&1&2&3&4&5&6&7 \\ 0&1&6&3&4&5&2&7 \end{pmatrix}$
P5 → P3	Pos. 5 ↔ Pos. 3	$p=\begin{pmatrix} 0&1&2&3&4&5&6&7 \\ 0&1&6&5&4&3&2&7 \end{pmatrix}$
P0 → P2	Pos. 0 ↔ Pos. 6	$p=\begin{pmatrix} 0&1&2&3&4&5&6&7 \\ 2&1&6&5&4&3&0&7 \end{pmatrix}$
P4 → P5	Pos. 4 ↔ Pos. 3	$p=\begin{pmatrix} 0&1&2&3&4&5&6&7 \\ 2&1&6&4&5&3&0&7 \end{pmatrix}$
wenn Datenübertragung von P4 → P5 abgeschlossen:		
P6 → P5	Pos. 4 ↔ Pos. 6	$p=\begin{pmatrix} 0&1&2&3&4&5&6&7 \\ 2&1&6&4&0&3&5&7 \end{pmatrix}$

Zusätzliches Bit zur Validierung der Zieladresse. Das Prinzip des Tauschens der Zieladresse ist als reine Softwarelösung mit Hilfe von Zeigerarithmetik und indirekter Adressierung von Datenfeldern sehr einfach und effektiv zu implementieren. Die Realisierung mit VHDL ist dagegen aufgrund der notwendigen Synthese in FPGA-Hardware sehr viel aufwändiger und wenig effektiv. Aus diesem Grund soll nun eine weitere Möglichkeit vorgestellt werden, wie das Routing im Schaltnetz trotz unvollständiger Permutationen mit FPGA-Hardware umgesetzt werden kann. Das Prinzip des Separation-Routings beruht im Wesentlichen darauf, in der linken Hälfte des Netzes komplementäre Zieladressen I und J mit $I(s)=J(s)=[i_{n-1}\ldots i_{s+1}]_2$ zu finden[47], um diese in verschiedene Teilnetze der nachfolgenden Stufe zu routen. Um nun das umständliche Tauschen zu umgehen, werden alle Zieladressen I um das Bit i_n erweitert, mit dem die Zieladresse als bestehender Kommunikationswunsch validiert wird. Die Zieladresse $I=[i_n\ldots i_0]_2$ wird vom Schaltnetz nur dann ausgewertet, wenn das Validierungsbit i_n gesetzt ist, d.h. wenn ein gültiger Verbindungswunsch besteht. Das Validierungsbit der Zieladresse wird nur gesetzt, wenn

[47] vgl. 5.5.5.3 Separation im linken Teil des Netzes, S. 125 ff.

eine Verbindung von einem Sender zu einem freien Empfänger der Nachricht aufgebaut werden soll. Bei mehrfach auftretenden Zieladressen (Gleichung 5.30) wird das Validierungsbit i_n nur für eine der gewünschten Verbindungen gesetzt, da zeitgleich nur eine Verbindung zu jedem Empfänger geroutet werden kann.

Routing der Beispielpermutation. Die **Abbildung 5.44** zeigt das Routing unvollständiger Abbildungen mit $n = 3$ am Beispiel der Permutation aus Gleichung 5.30. Die gültigen Verbindungswünsche der Sender 0, 2, 4 und 5 sind in der Abbildung fett dargestellt. Gleichzeitig wurden die Zieladressen wurden um das Validierungsbit i_3 erweitert, welches in den fett dargestellten Verbindungen gesetzt ist. Im Unterschied zum in der Abbildung 5.33 auf Seite 119 dargestellten Separation-Routing aus dem Richter-Netz werden nun in den einzelnen Stufen nur jene Zieladressen ausgewertet, bei denen das Validierungsbit i_3 gesetzt ist. Da somit alle anderen Zieladressen irrelevant werden, sind in der Abbildung 5.44 auch nur die signifikanten Adressbits der gültigen Verbindungswünsche dargestellt. Durch die Einführung des Validierungsbits i_3 wird es möglich, zu jeder Zeit einzelne Verbindungen zu einem freien Ausgang zu routen, ohne jedes Mal eine vollständige Permutation erzeugen zu müssen. Auf diese Weise eignet sich das Separation-Routing auch für alle praktischen Anwendungen, bei denen i.d.R. nicht alle Prozessoren zum gleichen Zeitpunkt an jeweils andere Empfänger senden möchten. Einzig die Frage nach einer z.B. prioritätsbasierten Arbitrationslogik für den Fall, dass zeitgleich mehrere Sender an den gleichen Empfänger senden möchten (im Beispiel P4 vor P6), soll an dieser Stelle offen bleiben.

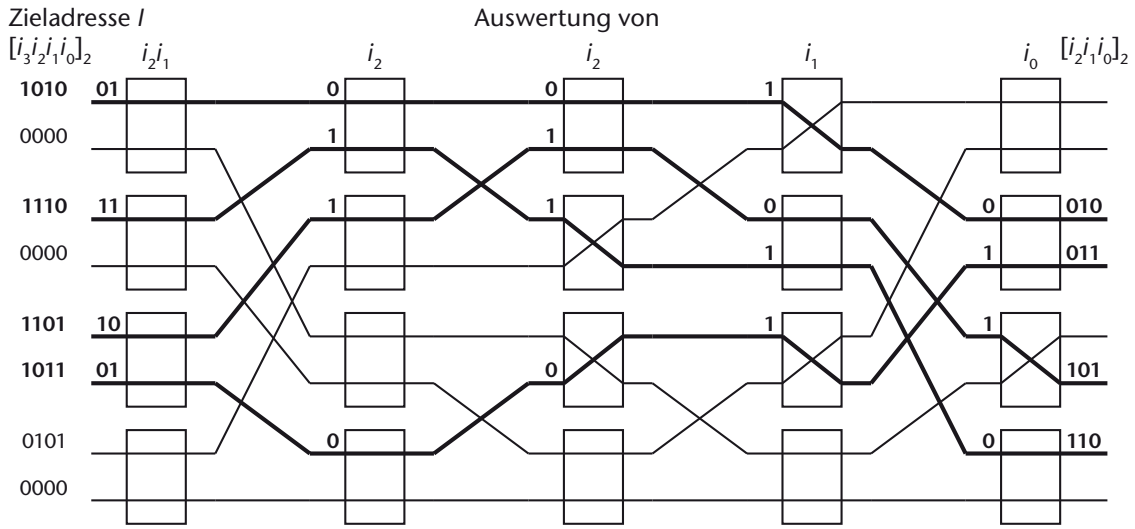


Abbildung 5.44: Separation-Routing unvollständiger Abbildungen

5.5.5.6 Routing in Boole'scher Algebra

Realisierung des Schaltnetzes. Nachdem die allgemeine Funktion des Schaltnetzes erläutert und der Nachweis über dessen korrekte Funktion erbracht wurde, soll nachfolgend gezeigt werden, wie das Schaltnetz mit Hilfe logischer Verknüpfungen von Zieladressen realisiert wird. Aus Gründen der geringeren Komplexität soll das Routing an einem Netzwerk der Größe $N = 8$ allgemein und exemplarisch am folgenden Verbindungswunsch dargestellt werden:

$$p = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & - & 6 & 1 & 5 & 3 & 5 & 0 \end{pmatrix} \qquad (5.31)$$

Testen auf Äquivalenz. Ob zwei Zieladressen komplementär sind, lässt sich in Boole'scher Algebra[48] durch eine Äquivalenz bzw. durch eine XNOR-Funktion testen. Hierzu werden die einzelnen Bits der binären Zieladressen miteinander verglichen. Sind alle Bits gleich (0 oder 1), so liefert die XNOR-Funktion eine 1 zurück. Im anderen Fall ist das Ergebnis eine 0. So sind zwei Zieladressen I_x und I_y komplementär, wenn $I_x(s) = I_y(s) = [i_{n-1} \ldots i_{s+1}]_2$ und beide Adressen gültig ($i_n = 1$) sind und damit die folgende Bedingung Y erfüllt ist:

$$Y = \overline{I_x(i_{n-1}) \veebar I_y(i_{n-1})} \wedge \ldots \wedge \overline{I_x(i_{s+1}) \veebar I_y(i_{s+1})} \wedge I_x(i_n) \wedge I_y(i_n) \qquad (5.32)$$

Routing der ersten Schalterstufe. Das Routing beginnt mit der ersten Schalterstufe $s = 0$. Wie in der **Abbildung 5.45** zu sehen ist, wird der Schalter S0.0 willkürlich auf die Stellung *gerade* (=) gesetzt. Dadurch entsteht bereits die erste Abhängigkeit im Netz. Alle weiteren Schalter dürfen nun keine komplementären Zieladressen von I'_0 und I'_1 auf denselben Ausgang routen, da diese in verschiedene Teilnetze geroutet werden müssen[49]. In der ersten Schalterstufe sind zwei Adressen I_x und I_y komplementär, wenn sie sich lediglich in dem niedrigstwertigen Bit i_0 unterscheiden, d.h. wenn $I_x(s) = I_y(s) = [i_{n-1} \ldots i_2 i_1]_2$. Wie im Abschnitt „5.5.5.3.2 Finden von Abhängigkeiten und Auflösen von Konflikten" auf Seite 130 ff. beschrieben, müssen im 1. Vergleich die Zieladressen der Eingänge jedes Schalters mit den Zieladressen der vorhergehenden Schalter der Stufe verglichen werden. Ist ein komplementäres Adressenpaar gefunden, so wird der Schalter mit S'_e gesetzt und mit f_e fixiert.

[48] George Boole (1815-1864), englischer Mathematiker, Logiker und Philosoph

[49] vgl. 5.5.4 Separation-Routing, S. 117 ff.

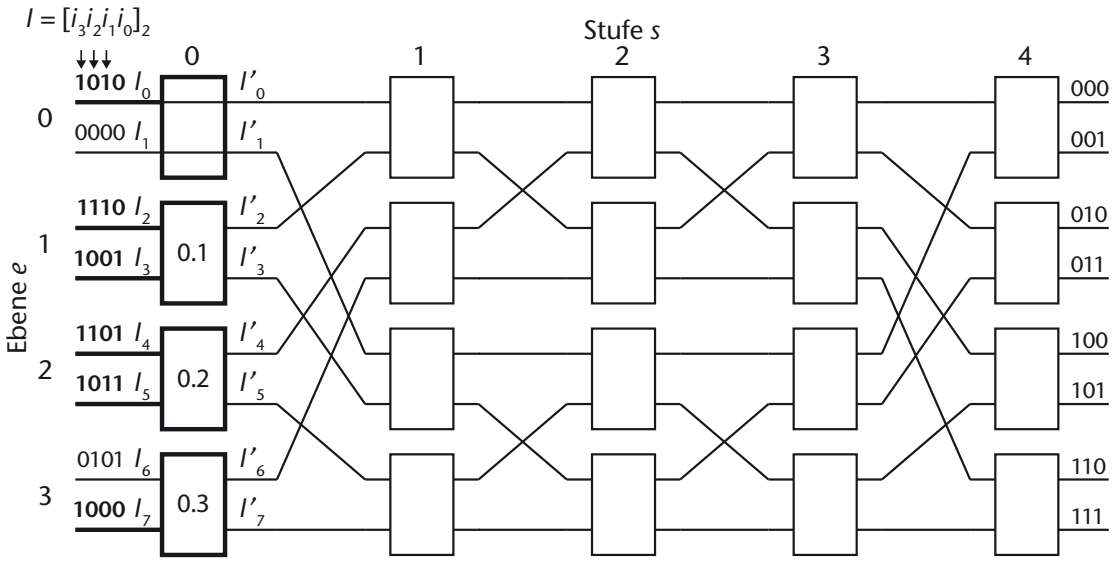


Abbildung 5.45: Beispielpermutation (Routing der Schalterstufe s = 0)

1. Vergleich: Adressen vergleichen. Die Gleichungen 5.33 bis 5.42 zeigen die Bestimmung von g und k der Stufe $s = 0$ im ersten Vergleich der Zieladressen, wie sie auch in der Tabelle 5.7 auf Seite 133 aufgeführt sind.

$$g_2 = \overline{I_2(i_2) \underline{\vee} I'_1(i_2)} \wedge \overline{I_2(i_1) \underline{\vee} I'_1(i_1)} \wedge I_2(i_3) \wedge I'_1(i_3) \tag{5.33}$$

$$g_3 = \overline{I_3(i_2) \underline{\vee} I'_0(i_2)} \wedge \overline{I_3(i_1) \underline{\vee} I'_0(i_1)} \wedge I_3(i_3) \wedge I'_0(i_3) \tag{5.34}$$

$$\begin{aligned} g_4 = \; & \left(\overline{I_4(i_2) \underline{\vee} I'_1(i_2)} \wedge \overline{I_4(i_1) \underline{\vee} I'_1(i_1)} \wedge I_4(i_3) \wedge I'_1(i_3) \right) \\ & \vee \left(\overline{I_4(i_2) \underline{\vee} I'_3(i_2)} \wedge \overline{I_4(i_1) \underline{\vee} I'_3(i_1)} \wedge I_4(i_3) \wedge I'_3(i_3) \wedge f_1 \right) \end{aligned} \tag{5.35}$$

$$\begin{aligned} g_5 = \; & \left(\overline{I_5(i_2) \underline{\vee} I'_0(i_2)} \wedge \overline{I_5(i_1) \underline{\vee} I'_0(i_1)} \wedge I_5(i_3) \wedge I'_0(i_3) \right) \\ & \vee \left(\overline{I_5(i_2) \underline{\vee} I'_2(i_2)} \wedge \overline{I_5(i_1) \underline{\vee} I'_2(i_1)} \wedge I_5(i_3) \wedge I'_2(i_3) \wedge f_1 \right) \end{aligned} \tag{5.36}$$

$$k_2 = \overline{I_2(i_2) \underline{\vee} I'_0(i_2)} \wedge \overline{I_2(i_1) \underline{\vee} I'_0(i_1)} \wedge I_2(i_3) \wedge I'_0(i_3) \tag{5.37}$$

$$k_3 = \overline{I_3(i_2) \underline{\vee} I'_1(i_2)} \wedge \overline{I_3(i_1) \underline{\vee} I'_1(i_1)} \wedge I_3(i_3) \wedge I'_1(i_3) \tag{5.38}$$

$$\begin{aligned} k_4 = \quad & \left(\overline{I_4(i_2) \underline{\vee} I_0'(i_2)} \wedge \overline{I_4(i_1) \underline{\vee} I_0'(i_1)} \wedge I_4(i_3) \wedge I_0'(i_3)\right) \\ & \vee \left(\overline{I_4(i_2) \underline{\vee} I_2'(i_2)} \wedge \overline{I_4(i_1) \underline{\vee} I_2'(i_1)} \wedge I_4(i_3) \wedge I_2'(i_3) \wedge f_1\right) \end{aligned} \tag{5.39}$$

$$\begin{aligned} k_5 = \quad & \left(\overline{I_5(i_2) \underline{\vee} I_1'(i_2)} \wedge \overline{I_5(i_1) \underline{\vee} I_1'(i_1)} \wedge I_5(i_3) \wedge I_1'(i_3)\right) \\ & \vee \left(\overline{I_5(i_2) \underline{\vee} I_3'(i_2)} \wedge \overline{I_5(i_1) \underline{\vee} I_3'(i_1)} \wedge I_5(i_3) \wedge I_3'(i_3) \wedge f_1\right) \end{aligned} \tag{5.40}$$

$$\begin{aligned} k_6 = \quad & \left(\overline{I_6(i_2) \underline{\vee} I_0'(i_2)} \wedge \overline{I_6(i_1) \underline{\vee} I_0'(i_1)} \wedge I_6(i_3) \wedge I_0'(i_3)\right) \\ & \vee \left(\overline{I_6(i_2) \underline{\vee} I_2'(i_2)} \wedge \overline{I_6(i_1) \underline{\vee} I_2'(i_1)} \wedge I_6(i_3) \wedge I_2'(i_3) \wedge f_1\right) \\ & \vee \left(\overline{I_6(i_2) \underline{\vee} I_4'(i_2)} \wedge \overline{I_6(i_1) \underline{\vee} I_4'(i_1)} \wedge I_6(i_3) \wedge I_4'(i_3) \wedge f_2\right) \end{aligned} \tag{5.41}$$

$$\begin{aligned} k_7 = \quad & \left(\overline{I_7(i_2) \underline{\vee} I_1'(i_2)} \wedge \overline{I_7(i_1) \underline{\vee} I_1'(i_1)} \wedge I_7(i_3) \wedge I_1'(i_3)\right) \\ & \vee \left(\overline{I_7(i_2) \underline{\vee} I_3'(i_2)} \wedge \overline{I_7(i_1) \underline{\vee} I_3'(i_1)} \wedge I_7(i_3) \wedge I_3'(i_3) \wedge f_1\right) \\ & \vee \left(\overline{I_7(i_2) \underline{\vee} I_5'(i_2)} \wedge \overline{I_7(i_1) \underline{\vee} I_5'(i_1)} \wedge I_7(i_3) \wedge I_5'(i_3) \wedge f_2\right) \end{aligned} \tag{5.42}$$

1. Vergleich: Schalter setzen. Nun können die Schalter der Stufe $s = 0$ im ersten Vergleich mit den Gleichungen 5.43 bis 5.47 gesetzt und fixiert werden.

$$S_1' = k_2 \vee k_3 \tag{5.43}$$

$$S_2' = k_4 \vee k_5 \tag{5.44}$$

$$S_3' = k_6 \vee k_7 \tag{5.45}$$

$$f_1 = g_2 \vee g_3 \vee k_2 \vee k_3 \tag{5.46}$$

$$f_2 = g_4 \vee g_5 \vee k_4 \vee k_5 \tag{5.47}$$

Die **Tabelle 5.26** zeigt das Ergebnis des 1. Vergleichs für die vier Schalter S0.0 bis S0.3 der ersten Stufe.

Tabelle 5.26: Beispielpermutation (Routing der Schalterstufe s = 0 im 1. Vergleich)

e	m	I_m		Schalterstellung, 1. Vergleich							
		gültig	Ziel	i_n	$I(s)$	g	k	f_e	S'_e	i'_n	$I'(s)$
0	0	1	2	1	01	-	-	1	0	1	01
	1	0	0	0	00	-	-			0	00
1	2	1	6	1	11	0	0	0	0	1	11
	3	1	1	1	00	0	0			1	00
2	4	1	5	1	10	0	0	1	0	1	10
	5	1	3	1	01	1	0			1	01
3	6	0	5	0	10	-	0	-	1	1	00
	7	1	0	1	00	-	1			0	10

2. Vergleich: Adressen vergleichen. Nach dem 1. Vergleich werden im 2. Vergleich wieder die Zieladressen der Eingänge mit den Ausgängen der Schalter verglichen, dieses Mal jedoch in der umgekehrten Reihenfolge und nur für die Stellung *gekreuzt* und unter Berücksichtigung des 1. Vergleichs (vgl. Tabelle 5.9 auf Seite 136). Die Gleichungen 5.48 bis 5.51 zeigen den 2. Vergleich für die Stufe $s = 0$.

$$k'_5 = \overline{I'_5(i_2) \underline{\vee} I''_7(i_2)} \wedge \overline{I'_5(i_1) \underline{\vee} I''_7(i_1)} \wedge I'_5(i_3) \wedge I''_7(i_3) \tag{5.48}$$

$$k'_4 = \overline{I'_4(i_2) \underline{\vee} I''_6(i_2)} \wedge \overline{I'_4(i_1) \underline{\vee} I''_6(i_1)} \wedge I'_4(i_3) \wedge I''_6(i_3) \tag{5.49}$$

$$\begin{aligned} k'_3 = \quad & \left(\overline{I'_3(i_2) \underline{\vee} I''_7(i_2)} \wedge \overline{I'_3(i_1) \underline{\vee} I''_7(i_1)} \wedge I'_3(i_3) \wedge I''_7(i_3) \right) \\ & \vee \left(\overline{I'_3(i_2) \underline{\vee} I''_5(i_2)} \wedge \overline{I'_3(i_1) \underline{\vee} I''_5(i_1)} \wedge I'_3(i_3) \wedge I''_5(i_3) \right) \end{aligned} \tag{5.50}$$

$$\begin{aligned} k'_2 = \quad & \left(\overline{I'_2(i_2) \underline{\vee} I''_6(i_2)} \wedge \overline{I'_2(i_1) \underline{\vee} I''_6(i_1)} \wedge I'_2(i_3) \wedge I''_6(i_3) \right) \\ & \vee \left(\overline{I'_2(i_2) \underline{\vee} I''_4(i_2)} \wedge \overline{I'_2(i_1) \underline{\vee} I''_4(i_1)} \wedge I'_2(i_3) \wedge I''_4(i_3) \right) \end{aligned} \tag{5.51}$$

Schalter setzen (2. Vergleich). Nun können die endgültigen Schalterstellungen der ersten Stufe mit den Gleichungen 5.52 bis 5.54 ermittelt werden.

$$S_1 = S'_1 \vee k'_2 \vee k'_3 \tag{5.52}$$

$$S_2 = S'_2 \vee k'_4 \vee k'_5 \tag{5.53}$$

$$S_3 = S'_3 \tag{5.54}$$

Die **Tabelle 5.27** zeigt die ermittelten Schalterstellungen der ersten Stufe $s = 0$.

Tabelle 5.27: Beispielpermutation (Routing der Schalterstufe s = 0 im 2. Vergleich)

e	m	Eingang I_m		Schalterstellung, 2. Vergleich						Ausgang I'_m	
		gültig	Ziel	i'_n	$I'(s)$	k'	S_e	i''_n	$I''(s)$	gültig	Ziel
0	0	1	2	1	01	-	0	1	01	1	2
	1	0	0	0	00	-		0	00	0	0
1	2	1	6	1	11	0	0	1	11	1	6
	3	1	1	1	00	0		1	00	1	1
2	4	1	5	1	10	0	0	1	10	1	5
	5	1	3	1	01	0		1	01	1	3
3	6	0	5	1	00	-	1	1	00	1	0
	7	1	0	0	10	-		0	10	0	5

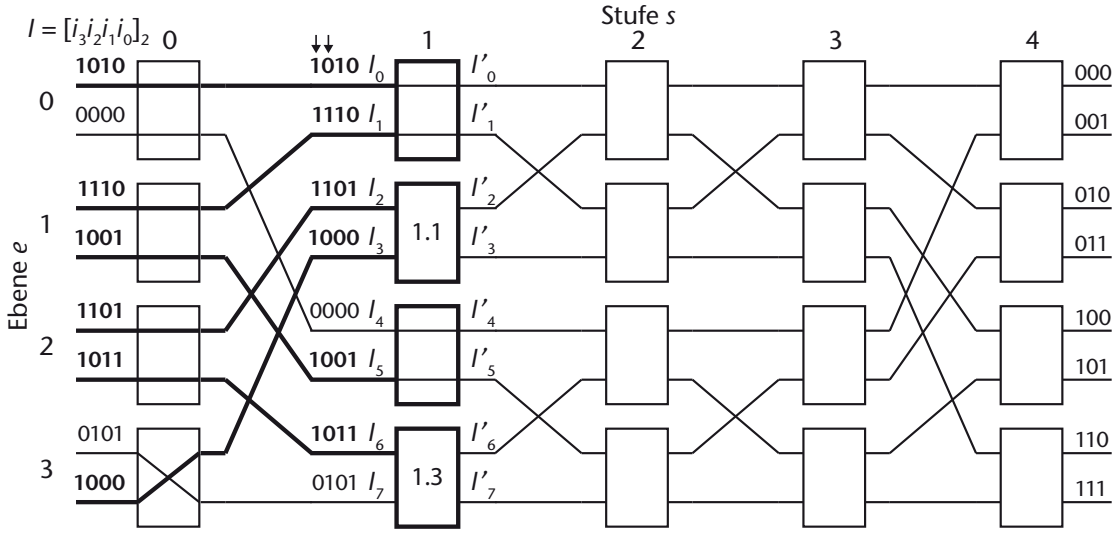


Abbildung 5.46: Beispielpermutation (Routing der Schalterstufe s = 1)

Routing der zweiten Schalterstufe. In der zweiten Schalterstufe $s = 1$ wird das Routing vereinfacht, indem sich die Auswertung der Zieladressen auf das höchstwertige Bit i_2 beschränkt. Zudem wird, wie auch in der ersten Schalterstufe, die erste Schalterstellung als *gerade* festgelegt. Aufgrund der Aufteilung in zwei unabhängige Teilnetze wird in der zweiten Schalterstufe neben dem Schalter S1.0 auch der Schalter S1.2 auf die Stellung *gerade* gesetzt (vgl. **Abbildung 5.46**). Anschließend erfolgt das Setzen der Schalter S1.1 und S1.3. Da beide Schalter nur jeweils von einem vorhergehenden Schalter abhängig sind, entfällt der zweite Vergleich. Da mit S1.0 und S1.2 auch die Schalterstellung der vorhergehenden Schalter feststeht, müssen für die Bestimmung der Schalterstellung von S1.1 und S1.3 lediglich die Zieladressen der Eingänge miteinander verglichen werden:

$$S_1 = \left(\overline{I_2(i_2) \veebar I_0(i_2)} \wedge I_2(i_3) \wedge I_0(i_3)\right) \vee \left(\overline{I_3(i_2) \veebar I_1(i_2)} \wedge I_3(i_3) \wedge I_1(i_3)\right) \quad (5.55)$$

$$S_3 = \left(\overline{I_6(i_2) \veebar I_4(i_2)} \wedge I_6(i_3) \wedge I_4(i_3)\right) \vee \left(\overline{I_7(i_2) \veebar I_5(i_2)} \wedge I_7(i_3) \wedge I_5(i_3)\right) \quad (5.56)$$

Das Routing der linken $\log_2 N$-1 Schalterstufen ist damit abgeschlossen. Die **Abbildung 5.47** zeigt die fertig gerouteten Schalter sowie die daraus resultierende Permutation der Zieladressen am Eingang der Mittelstufe. Wie zu erkennen ist, befinden sich keine komplementären Adressen I_x und I_y mit $I_x(i_{n-1}\ldots i_1) = I_y(i_{n-1}\ldots i_1)$ in derselben Hälfte. Darüber hinaus unterscheiden sich zwei gültige Adressen I_x und I_{x+1} mit $x = \{0,2,\ldots,N\text{-}2\}$ am Eingang jedes Schalters der Mittelstufe in dem höchstwertigen Bit i_2, was ebenfalls ein Resultat aus dem Separationsteil ist.

Tabelle 5.28: Beispielpermutation (Routing der Schalterstufe s = 1)

e	**m**	**Eingang I_m**		**Vergleich**					**Ausgang I'_m**	
		gültig	**Ziel**	i_n	$I(s)$	S_e	i'_n	$I'(s)$	**gültig**	**Ziel**
0	0	1	2	1	0	0	1	0	1	2
	1	1	6	1	1		1	1	1	6
1	2	1	5	1	1	0	1	1	1	5
	3	1	0	1	0		1	0	1	0
2	4	0	0	0	0	0	0	0	0	0
	5	1	1	1	0		1	0	1	1
3	6	1	3	1	0	0	1	0	1	3
	7	0	5	0	1		0	1	0	5

Routing der rechten $\log_2 N$ Schalterstufen. Das Routing der noch fehlenden rechten $\log_2 N$ Stufen erfolgt analog wie in einem konventionellen inversen Baseline-Netz[50] durch die bitweise Auswertung der binären Zieladresse. Wie die **Abbildung 5.47** bis **Abbildung 5.49** zeigt, wird neben dem Validierungsbit (i_n) in jeder Stufe ein einzelnes Bit ausgewertet, angefangen mit dem MSB (i_{n-1}) in der Mittelstufe bis zum LSB (i_0) in der Ausgangsstufe. Steht an einem Eingang mit einer gültigen Adresse im auszuwertenden Bit *i* eine *Null*, dann wird dieser Eingang in den obere Ausgang geroutet. Im anderen Fall, wenn an einem Eingang mit einer gültigen Adresse im auszuwertenden Bit *i* eine *Eins* steht, dann wird dieser Eingang in den unteren Ausgang geroutet. Aufgrund der Vorsortierung im Separationsteil tritt immer der Fall ein, dass im auszuwertenden Bit *i* in einer Zieladresse eine *Null* und in der Zieladresse am anderen Eingang desselben Schalters eine *Eins* steht, so dass prinzi-

[50] beschrieben in [Ri97, S. 179 f.]

piell nur ein Eingang ausgewertet werden muss. Da es bei unvollständigen Permutationen jedoch möglich ist, dass eine der beiden Zieladressen an den Eingängen des Schalter ungültig ist, müssen im Unterschied zum konventionellen Routing im Baseline-Netz immer beide Zieladressen ausgewertet werden[51]. Für die Schalter der rechten $\log_2 N$ Stufen wird die Schalterstellung durch

$$S_e = \left(I_{2e}(i_{2n-1-s}) \wedge I_{2e}(i_n)\right) \vee \left(\overline{I_{2e+1}(i_{2n-1-s})} \wedge I_{2e+1}(i_n)\right) \qquad (5.57)$$

ermittelt, mit $e = \{0,1,\ldots,(N/2)-1\}$ als Nummer der Ebene, $s = \{n, n+1,\ldots,2n-2\}$ als Nummer der Stufe und $n = \log_2 N$.

Routing der mittleren Schalterstufe. Die **Abbildung 5.47** zeigt die Zieladressen sowie die auszuwertenden Bits der Zieladressen (durch Pfeil markiert) an den Eingängen der dritten Schalterstufe $s = 2$.

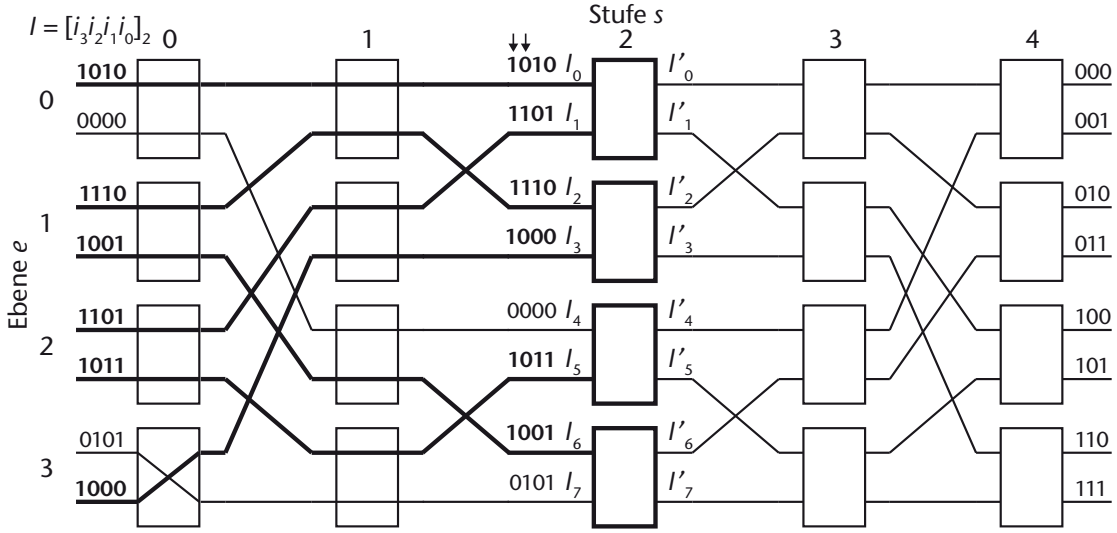


Abbildung 5.47: Beispielpermutation (Routing der Schalterstufe s = 2)

In der Beispielpermutation gelten mit Gleichung 5.57 für die mittlere Stufe $s = 2$ die folgenden Schalterstellungen:

$$S_0 = \left(I_0(i_2) \wedge I_0(i_3)\right) \vee \left(\overline{I_1(i_2)} \wedge I_1(i_3)\right) \qquad (5.58)$$

$$S_1 = \left(I_2(i_2) \wedge I_2(i_3)\right) \vee \left(\overline{I_3(i_2)} \wedge I_3(i_3)\right) \qquad (5.59)$$

$$S_2 = \left(I_4(i_2) \wedge I_4(i_3)\right) \vee \left(\overline{I_5(i_2)} \wedge I_5(i_3)\right) \qquad (5.60)$$

[51] vgl. Tabelle 5.24: Routing im Approximationsteil, S. 146

$$S_3 = \left(I_6(i_2) \wedge I_6(i_3)\right) \vee \left(\overline{I_7(i_2)} \wedge I_7(i_3)\right) \qquad (5.61)$$

Die **Tabelle 5.29** zeigt das Routing der Beispielpermutation aus Abbildung 5.47 in der mittleren Schalterstufe $s = 2$.

Tabelle 5.29: Beispielpermutation (Routing der Schalterstufe s = 2)

e	**m**	**Eingang I_m**		**Vergleich**			**Ausgang I'_m**	
		gültig	**Ziel**	i_n	$I(s)$	S_e	**gültig**	**Ziel**
0	0	1	2	1	0	0	1	2
	1	1	5	1	1		1	5
1	2	1	6	1	1	1	1	0
	3	1	0	1	0		1	6
2	4	0	0	0	0	1	0	3
	5	1	3	1	0		1	0
3	6	1	1	1	0	0	1	1
	7	0	5	0	1		0	5

Routing der vorletzten Schalterstufe. Die **Abbildung 5.48** zeigt die Zieladressen sowie die auszuwertenden Bits der Zieladressen an den Eingängen der vierten Schalterstufe $s = 3$.

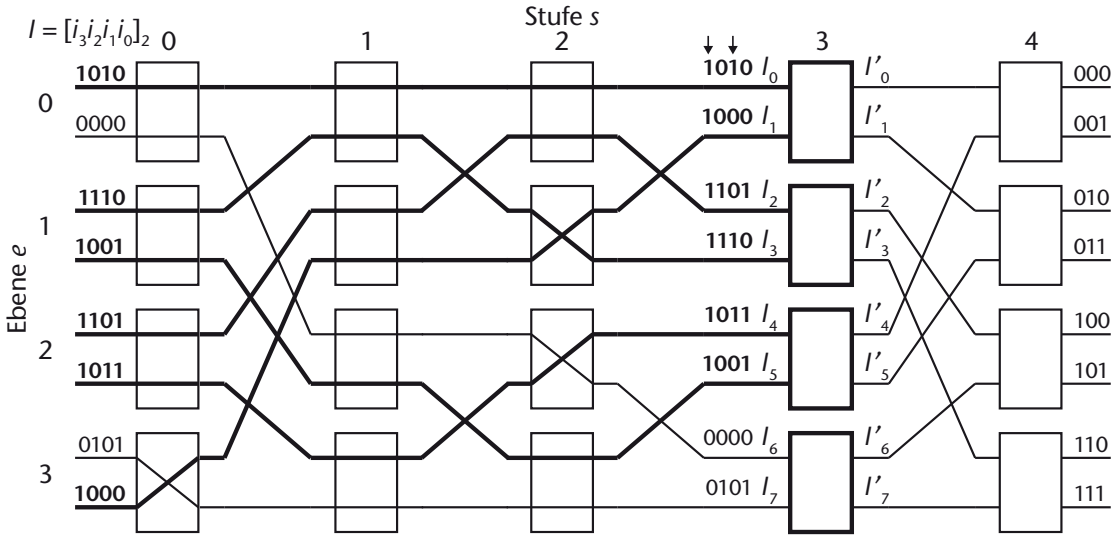


Abbildung 5.48: Beispielpermutation (Routing der Schalterstufe s = 3)

In der Beispielpermutation gelten mit Gleichung 5.57 für die vorletzte Stufe $s = 3$ die folgenden Schalterstellungen:

$$S_0 = \left(I_0(i_1) \wedge I_0(i_3)\right) \vee \left(\overline{I_1(i_1)} \wedge I_1(i_3)\right) \qquad (5.62)$$

$$S_1 = \left(I_2(i_1) \wedge I_2(i_3)\right) \vee \left(\overline{I_3(i_1)} \wedge I_3(i_3)\right) \tag{5.63}$$

$$S_2 = \left(I_4(i_1) \wedge I_4(i_3)\right) \vee \left(\overline{I_5(i_1)} \wedge I_5(i_3)\right) \tag{5.64}$$

$$S_3 = \left(I_6(i_1) \wedge I_6(i_3)\right) \vee \left(\overline{I_7(i_1)} \wedge I_7(i_3)\right) \tag{5.65}$$

Die **Tabelle 5.30** zeigt das Routing der Beispielpermutation aus Abbildung 5.48 in der vorletzten Schalterstufe $s = 3$.

Tabelle 5.30: Beispielpermutation (Routing der Schalterstufe s = 3)

e	m	Eingang I_m		Vergleich			Ausgang I'_m	
		gültig	Ziel	i_n	$I(s)$	S_e	gültig	Ziel
0	0	1	2	1	1	1	1	0
	1	1	0	1	0		1	2
1	2	1	5	1	0	0	1	5
	3	1	6	1	1		1	6
2	4	1	3	1	1	1	1	1
	5	1	1	1	0		1	3
3	6	0	0	0	0	0	0	0
	7	0	5	0	0		0	5

Routing der letzten Schalterstufe. Die **Abbildung 5.49** zeigt die Zieladressen sowie die auszuwertenden Bits der Zieladressen an den Eingängen der fünften Schalterstufe $s = 4$.

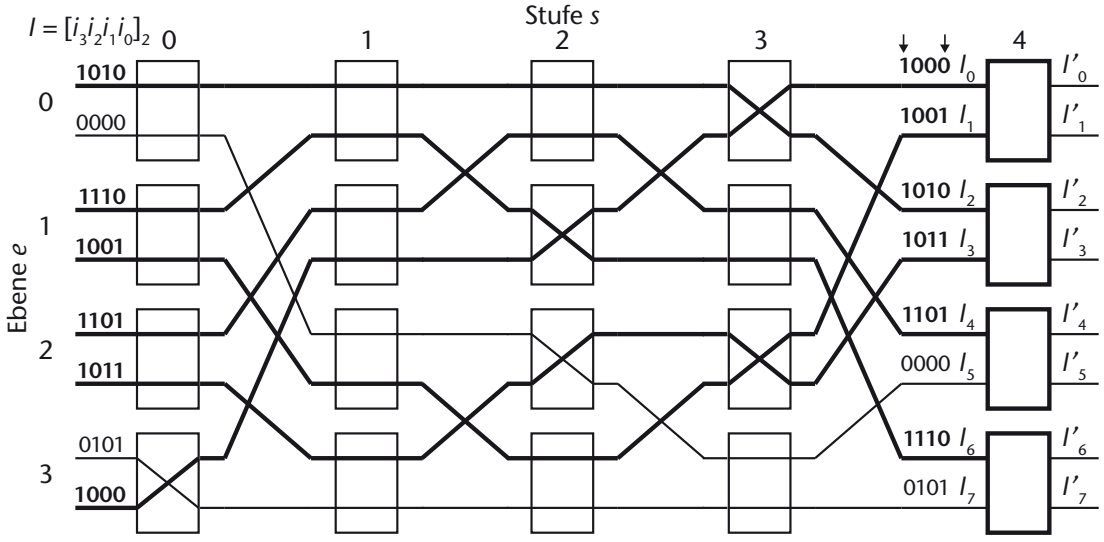


Abbildung 5.49: Beispielpermutation (Routing der Schalterstufe s = 4)

In der Beispielpermutation gelten mit Gleichung 5.57 für die letzte Stufe $s = 4$ die folgenden Schalterstellungen:

$$S_0 = \left(I_0(i_0) \wedge I_0(i_3)\right) \vee \left(\overline{I_1(i_0)} \wedge I_1(i_3)\right) \tag{5.66}$$

$$S_1 = \left(I_2(i_0) \wedge I_2(i_3)\right) \vee \left(\overline{I_3(i_0)} \wedge I_3(i_3)\right) \tag{5.67}$$

$$S_2 = \left(I_4(i_0) \wedge I_4(i_3)\right) \vee \left(\overline{I_5(i_0)} \wedge I_5(i_3)\right) \tag{5.68}$$

$$S_3 = \left(I_6(i_0) \wedge I_6(i_3)\right) \vee \left(\overline{I_7(i_0)} \wedge I_7(i_3)\right) \tag{5.69}$$

Die **Tabelle 5.31** zeigt das Routing der Beispielpermutation aus Abbildung 5.49 in der letzten Schalterstufe $s = 4$.

Tabelle 5.31: Beispielpermutation (Routing der Schalterstufe s = 4)

e	**m**	**Eingang I_m**		**Vergleich**			**Ausgang I'_m**	
		gültig	**Ziel**	i_n	$I(s)$	S_e	**gültig**	**Ziel**
0	0	1	0	1	0	0	1	0
	1	1	1	1	1		1	1
1	2	1	2	1	0	0	1	2
	3	1	3	1	1		1	3
2	4	1	5	1	1	1	0	0
	5	0	0	0	0		1	5
3	6	1	6	1	0	0	1	6
	7	0	5	0	1		0	5

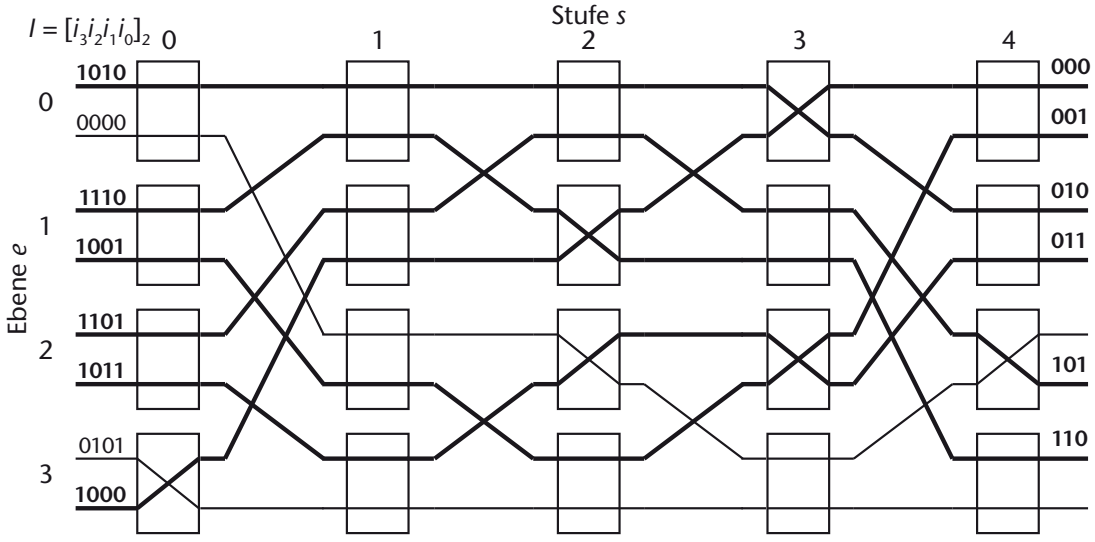


Abbildung 5.50: Beispielpermutation (Routing komplett)

5.5.5.7 Kosten in Hardware

Kleinste Erweiterung. In den Metriken eines Netzes gibt die kleinste Erweiterung eine Information darüber, wie das Netz mit zusätzlichen Teilnehmern erweitert werden kann, ohne die Topologie zu verändern. Aufgrund seines rekursiven Aufbaus[52] kann das Beneš-Netz nur durch eine Verdopplung der Teilnehmerzahl *N* erweitert werden. Wie die **Abbildung 5.51** zeigt, wird das Netz dupliziert und jeweils eine zusätzliche Schalterstufe am Eingang und am Ausgang hinzugefügt.

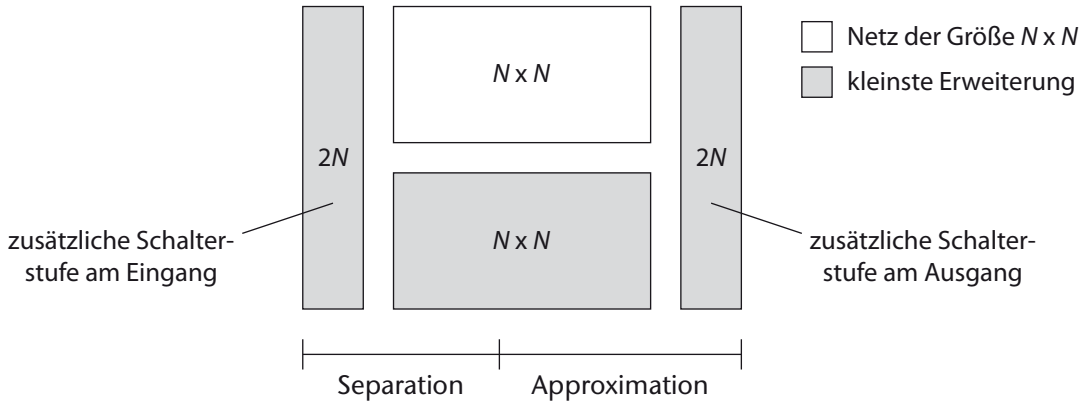


Abbildung 5.51: kleinste Erweiterung im Beneš-Netz

Anzahl der Schalter im Netz. Die Anzahl der Schalter definiert eine weitere wichtige Metrik des Netzes. Bei $2\log_2 N$-1 Stufen und $N/2$ Schalter pro Stufe sind

$$\frac{N}{2}\left(2\log_2 N - 1\right) \qquad (5.70)$$

Schalter im Netz enthalten (vgl. **Tabelle 5.32)**, deren Zieladressen am Schaltereingang durch das Schaltnetz ausgewertet werden müssen.

Tabelle 5.32: Anzahl der Schalter im Netz in Abhängigkeit der Netzgröße

Netzgröße (*N*)	Separationsteil	Approximationsteil	Gesamt
4	2	4	6
8	8	12	20
16	24	32	56
32	64	80	144
64	160	192	352
128	384	448	832
256	896	1024	1920
512	2048	2304	4352

[52] vgl. Abbildung 5.24: Rekursiver Aufbau des Beneš-Netzes, S. 105

Aufwand im rechten Teil des Netzes. In den rechten $\log_2 N$ Schalterstufen des Netzes besteht keine Abhängigkeit zwischen den Schaltern jeder Stufe. Aus diesem Grund steigen die Hardwarekosten für das Routing im Approximationsteil linear mit der Anzahl Schalter an (vgl. Spalte *Approximationsteil* in Tabelle 5.32).

Aufwand im linken Teil des Netzes. Anders als im Approximationsteil verhält es sich in den linken $\log_2 N$-1 Stufen des Netzes. Da hier die Schalter jedes Teilnetzes in jeder Stufe Abhängigkeiten zueinander aufweisen, steigen die Hardwarekosten für das Routing nichtlinear mit der Anzahl der Schalter im Netz an. So wird im ersten Vergleich jeder Eingang mit jedem Ausgang der darüberliegenden Schalter der Stufe verglichen (Ermittlung von g und k). Im zweiten Vergleich wird jeder Eingang mit jedem zweiten Ausgang der darunterliegenden Schaltern verglichen (Ermittlung von k). So sind bei $N(s) \geq 8$ in jeder Stufe insgesamt

$$k = 2^s \left(\sum_{e=1}^{N(s)/2-1} 4e + \sum_{e=1}^{N(s)/2-2} 2e \right) \qquad (5.71)$$

einzelne Vergleiche nötig, mit $N(s) = N/2^s$ als Anzahl der Eingänge pro Teilnetz in einer Stufe und $s = \{0,1,\ldots,\log_2 N - 1\}$ als die Nummer der Stufe.

Tabelle 5.33: Hardwarekosten im linken Teil des Netzes

Anzahl der Vergleiche für jedes Teilnetz		**Anzahl der Vergleiche im Separationsteil des Netzes**	
$N(s)$	k mit $s = 0$	N	k mit $s = \{0,1,\ldots,\log_2 N\text{-}1\}$
4	2	4	2
8	30	8	34
16	154	16	222
32	690	32	1134
64	2914	64	5182
128	11970	128	22334
256	48514	256	93182
512	195840	512	382204

Fazit. Wie die **Tabelle 5.33** zeigt, steigen die Hardwarekosten des Schaltnetzes aufgrund der hohen Schalterzahl (vgl. Tabelle 5.32) und dem Prinzip der adressvergleichenden Auswertung im Schaltnetz in den linken Stufen des Netzes gegenüber der Teilnehmerzahl N sehr stark an. Auf der anderen Seite kann das Netz weit unterhalb einer Taktzeit geroutet werden, sofern die für das Schaltnetz benötigten Ressourcen auf dem Chip zur Verfügung stehen.

5.5.6 Implementierung im FPGA

Leitungsvermittlung. Die prototypische Implementierung der Interprozessorkomunikation bei *ConPar* erfolgt als leitungsvermittelndes Netzwerk (NoC) in FPGA-Hardware. Das bedeutet, dass im Netzwerk eine direkte Verbindung zwischen Sender und Empfänger der Nachricht vermittelt wird, so dass die Nachricht ohne Zwischenknoten direkt übertragen werden kann und auch nicht an irgendeiner Stelle (z.B. den Knotenpunkten) im Netz zwischengespeichert werden muss. Wie die **Abbildung 5.52** am Beispiel unidirektionaler FSL-Verbindungen[53] zeigt, erfolgt die Übertragung der Daten über die Vermittlung von Signalen vom Sender zum Empfänger entsprechend der Topologie des Netzes. Aufgrund der Leitungsvermittlung reduziert sich die Latenz der Datenübertragung auf die reine Gatterlaufzeit der FPGA-Hardware, was einige Größenordnungen unterhalb einer Taktperiode liegt, weshalb mit jedem Taktzyklus ein neuer Wert direkt vom Sender zum Empfänger übertragen werden kann. Das leitungsvermittelnde Netz selbst ist für beide Seiten transparent. Die Gatterlaufzeit steigt mit der Netzgröße analog zur Stufenzahl mit $O\left(2\log_2 N-1\right)$, liegt aber bei realistischen Teilnehmerzahlen von $N \leq 512$ noch immer weit unterhalb der Taktzeit.

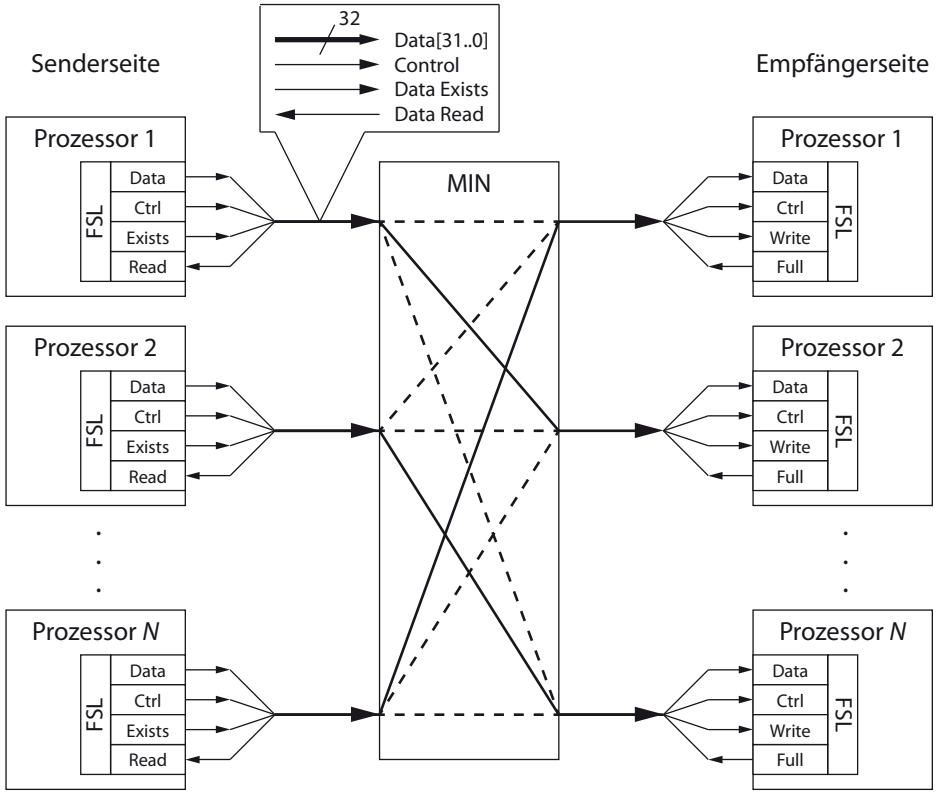


Abbildung 5.52: Leitungsvermittlung von FSL-Verbindungen im FPGA

[53] Fast Simplex Link, kurz FSL [vgl. Xi08b, S. 83 f.]

5.5.6.1 Verdrahtungsstufen

Zuweisung von Signalen. Die Leitungsvermittlung im Netzwerk basiert auf der Verkettung von Signalen, welche in diesem Fall die Datenleitungen darstellen. Verdrahtungsstufen bilden im Netzwerk eine feste Abbildung (Permutation) der Eingänge auf die Ausgänge der jeweiligen Stufe. Die **Abbildung 5.53** zeigt als Beispiel eine Verdrahtungsstufe der Größe 2 x 2 mit der dazugehörigen Darstellung der Shuffle-Permutation.

$$\sigma_1 = \begin{pmatrix} A & B \\ D & C \end{pmatrix}$$

Abbildung 5.53: Verdrahtungsstufe mit n = 1

Unbedingte Signalzuweisung in VHDL. Die Permutation einer Verdrahtungsstufe lässt sich in VHDL außerhalb von Prozessen sehr einfach durch eine *unbedingte Signalzuweisung* realisieren. Im **Listing 5.1** ist die Implementierung der Verdrahtungsstufe aus Abbildung 5.53 als unbedingte Zuweisung von zwei 32-Bit breiten Signalvektoren in VHDL-Code aufgeführt.

Listing 5.1: Verdrahtungsstufe mit n = 1 in VHDL

```
-- 2x2 Verdrahtung
signal A, B, C, D : std_logic_vector[0 to 31];
C <= B;
D <= A;
```

5.5.6.2 Kreuzschalter

Dynamische Verkettung von Signalen. Die **Abbildung 5.54** zeigt einen Kreuzschalter der Dimension 2 x 2, wie er als Schaltelement im dynamischen Verbindungsnetzwerk verwendet wird. Jeder 2 x 2-Kreuzschalter besitzt zwei Eingänge (*A*, *B*) und zwei Ausgänge (*C*, *D*). Wie in der Abbildung 5.54 dargestellt, kann ein Kreuzschalter die zwei Zustände *gerade* ($S = 0$) oder *gekreuzt* ($S = 1$) einnehmen. Im Unterschied zu den Verdrahtungsstufen können die Ein- und Ausgänge der Schalterstufen dynamisch miteinander verkettet werden.

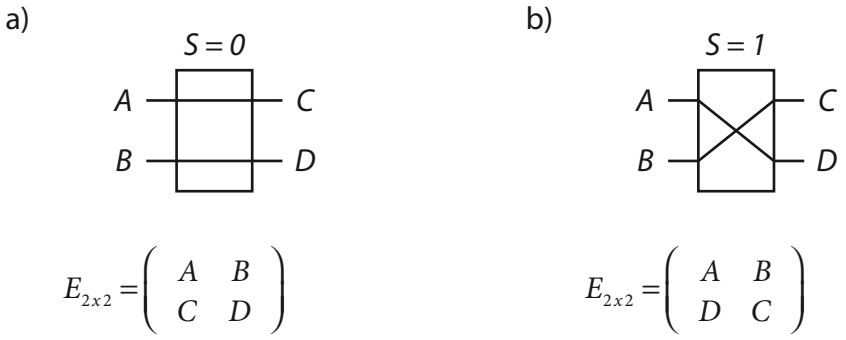


Abbildung 5.54: Schalterstellung a) gerade, b) gekreuzt

Selektive Signalzuweisung in VHDL. Der entscheidende Punkt bei der Implementierung von Kreuzschaltern in FPGA-Hardware ist die dynamische Zuweisung von verketteten Signalen. Ein 2 x 2-Kreuzschalter, wie in der Abbildung 5.54 dargestellt, lässt sich in VHDL-Code außerhalb von Prozessen durch eine *selektive Signalzuweisung* mit der Schalterstellung *S* als Parameter realisieren. Das **Listing 5.2** zeigt den Kreuzschalter aus Abbildung 5.54 als selektive Zuweisung von zwei 32-Bit breiten Signalvektoren in VHDL-Code.

Listing 5.2: 2x2 Kreuzschalter durch selektive Signalzuweisung in VHDL

```
-- 2x2 Kreuzschalter
signal A, B, C, D : std_logic_vector[0 to 31];
signal S : bit;
C <= A when S = '0' else B;
D <= A when S = '1' else B;
```

5.5.6.3 Routing

Simplifizierung der Boole'schen Algebra mit VHDL. Die Realisierung des Routings durch ein Schaltnetz[54] in FPGA-Hardware kann durch die logische Verknüpfung der Zieladressen an den Schalterein- und ausgängen in VHDL-Code erfolgen. Aufgrund der leistungsfähigen Funktionalität von VHDL gestaltet sich die Routing-Logik etwas eleganter und weniger aufwändig als bei der im Abschnitt 5.5.5.6 auf Seite 150 ff. gezeigten reinen Boole'schen Algebra. So kann im VHDL-Code der Vergleich einzelner Bits der Zieladressen durch den Vergleich von Bitvektoren deutlich vereinfacht werden. So sieht als Beispiel der VHDL-Code für den Schalter S0.1 aus Abbildung 5.45 auf Seite 151 wie folgt aus:

[54] vgl. 5.5.5.6 Routing in Boole'scher Algebra, S. 150 ff.

Listing 5.3: Routing von Schalter S1 im Separationsteil mit N(s) = 8 in VHDL

```
-- Deklaration von der Zieladressen als Bitvektor [i3,i2,i1,i0]
--   Schaltereingänge I
signal I0, I1, I2, I3,... : bit_vector (0 to 3);
--   Schalterausgänge I'
signal I0_1, I1_1, I2_1, I3_1,... : bit_vector (0 to 3);
--   Schalterausgänge I''
signal ..., I2_2, I3_2, I4_2, I5_2, I6_2, I7_2 : bit_vector (0 to 3);
-- Deklaration von g, k, f, S', k' und S als Bitsignal
signal g2, g3,... : bit;
signal k2, k3,... : bit;
signal f1,... : bit;
signal S1_1,... : bit;
signal S1,... : bit;
-- 1. Vergleich
  -- Bestimmung von g
  g2 <= '1' when ((I2(1 to 2) = I1_1(1 to 2)) and
                  (I2(0) = '1') and (I1_1(0) = '1')) else '0';
  g3 <= '1' when ((I3(1 to 2) = I0_1(1 to 2)) and
                  (I3(0) = '1') and (I0_1(0) = '1')) else '0';
  ...
  -- Bestimmung von k
  k2 <= '1' when ((I2(1 to 2) = I0_1(1 to 2)) and
                  (I2(0) = '1') and (I0_1(0) = '1')) else '0';
  k3 <= '1' when ((I3(1 to 2) = I1_1(1 to 2)) and
                  (I3(0) = '1') and (I1_1(0) = '1')) else '0';
  ...
   -- Bestimmung von S'
  S1_1 <= '1' when (k2 = '1') or (k3 = '1') else '0';
  ...
  -- Schalter markieren
  f1 <= '1' when (g2 = '1') or (g3 = '1') or
                  (k2 = '1') or (k3 = '1') else '0';
  ...
  -- Schalterausgänge setzen
  ...
  I2_1 <= I2 when S1_1 = '0' else I3;
  I3_1 <= I2 when S1_1 = '1' else I3;
...
-- 2. Vergleich
   -- Bestimmung von k
...
   k3_1 <= '1' when ((I3_1(1 to 2) = I7_2(1 to 2)) and
                     (I3_1(0) = '1') and (I7_2(0) = '1')) or
                     ((I3_1(1 to 2) = I5_2(1 to 2)) and
                     (I3_1(0) = '1') and (I5_2(0) = '1')) else '0';
```

```
   k2_1 <= '1' when ((I2_1(1 to 2) = I6_2(1 to 2)) and
                     (I2_1(0) = '1') and (I6_2(0) = '1')) or
                     ((I2_1(1 to 2) = I4_2(1 to 2)) and
                     (I2_1(0) = '1') and (I4_2(0) = '1')) else '0';
   -- Bestimmung von S
...
   S1 <= '1' when (S1_1 = '1') or (k2_1 = '1') or
                  (k3_1 = '1') else '0';
...
```

Schaltnetz in FPGA-Hardware. Eine detaillierte Beschreibung des Routingverfahrens in VHDL-Code ist im Anhang in „A.4.6 Routing“ auf Seite 261 ff. für ein Netzwerk der Größe 8 x 8 gegeben.

5.5.6.4 Übertragungsrate

32-Bit breite Verbindungen. Zur Datenübertragung wurde in der protoypischen Implementierung der FSL-Bus von Xilinx eingesetzt. Der FSL-Bus erlaubt eine 32-Bit breite Datenverbindung im Handshake-Verfahren, so dass 32 Bit Daten pro Taktzyklus übertragen werden[55]. Aufgrund der direkten Kommunikation zwischen Sender und Empfänger (Leitungsvermittlung) entsteht keine zusätzliche Latenz durch das Netz, weshalb die Netzgröße bei der Latenz keine Rolle spielt.

$$C_{\max} = 32Bit \cdot N \cdot f_{Netz} \qquad (5.72)$$

Bei einer Taktrate von f_{Netz} = 100 MHz im Virtex-4 FPGA beträgt die Übertragungsrate im Netz C = 3,2 GBit/s pro Verbindung. Im besten Fall, d.h. jeder Sender überträgt zeitgleich an einen anderen Empfänger, skaliert die maximale Übertragungsrate C_{max} für das gesamte Netz mit der Anzahl N der Teilnehmer (Gleichung 5.72). Bei N = 8 Teilnehmern bei gleichzeitiger Verwendung aller N Kanäle kann somit ein Datenvolumen von bis zu C_{max} = 25,6 GBit/s im Netz übertragen werden.

5.6 Zusammenfassung

Schlüsselfaktor Kommunikation. Die Interprozessorkommunikation im MPSoC stellt für den Echtzeitparallelrechner *ConPar* ein notwendiges Kriterium dar. Nur wenn die korrekte Übertragung von Echtzeitdaten zwischen vielen Prozessoren durch ein geeignetes Medium möglich ist, kann das Konzept von *Space-Sharing* realisiert werden.

[55] vgl. [Xi10, S. 5]

Statische Netze. In der Vergangenheit basierten Networks on-Chip (NoCs) auf Topologien statischer Netze, z.B. 2D-Gitter. Diese Topologien empfehlen sich jedoch aufgrund von *Multihops* und der Ausprägung von *Hotspots* nicht für die Übertragung von Echtzeitdaten in großen Netzen. Ein weiterer Nachteil besteht darin, dass jeder Rechenknoten gleichzeitig auch eine Routingfunktionalität im Netz übernehmen muss. Zudem erfordern statische Netze aus Gründen der Skalierbarkeit i.d.R. mehrdimensionale Netzwerkzugänge für jeden Knoten.

Dynamische Netze. Eine Alternative zu den statischen Netzen bieten dynamische Netze. Bei den dynamischen Netzen muss in zwei Kategorien, die nicht-blockierungsfreien Netze und die blockierungsfreien Netze, unterschieden werden. Beide Kategorien können als *leitungsvermittelnde Netze* betrieben werden und ermöglichen so, im Unterschied zu statischen Netzen, eine direkte Kommunikation zwischen Sender und Empfänger ohne Zwischenknoten. Ein weiterer Vorteil besteht darin, dass unabhängig von der Netzgröße für jeden Prozessor lediglich ein Netzwerkzugang nötig ist.

Nicht-blockierungsfreie Netze. Zu den bekanntesten Vertreter der nicht-blockierungsfreien Netze zählen die $\log_2 N$-Netze, wie z.B. das *Baseline-Netz*. $\text{Log}_2 N$-Netze bestehen aus Kreuzschaltern der Größe 2x2, welche in hintereinander geschalteten Stufen organisiert und durch Verdrahtungsstufen verbunden sind. Neben den sehr geringen Hardwarekosten bieten nicht-blockierungsfreie Netze den großen Vorteil äußerst einfacher Algorithmen für das Routen von Verbindungen durch das Netz. Das Routing kann dezentral auf Schalterebene erfolgen, so dass ein selbstorganisierendes Netz aus Kreuzschaltern entsteht. Nachteilig ist jedoch, dass zwar jeder Eingang mit jedem freien Ausgang, aber *nicht zu jeder Zeit* verbunden werden kann, da nur genau ein Pfad für jede mögliche Verbindung zur Verfügung steht (Pfadeindeutigkeit). Aus diesem Grund ist bei nicht-blockierungsfreien Netzen ein Scheduling der Verbindungswünsche in Form einer Schedulingtabelle nötig, um die Rechtzeitigkeit der Datenübertragung sicherzustellen. Dabei werden Nachrichten, welche zeitgleich übertragen werden können, zu gültigen Permutationen zusammengefasst. Diese Permutationen werden dann zur Laufzeit nach einem definierten Zeitplan als zeitliche Sequenz hergestellt. Der Vorteil dieses Verfahrens besteht, dass diese Methode ein vorhersagbares Kommunikationsverhalten und damit die Übertragung von Echtzeitdaten bei sehr geringem Hardwareaufwand erlaubt.

Blockierungsfreie Netze. Schaltet man zwei nicht-blockierungsfreie Netze hintereinander, so erhält man ein blockierungsfreies Netz wie das *Beneš*- oder das *Richter-Netz*, welches zu jedem Zeitpunkt jeden Eingang mit jedem freien Ausgang verbinden kann. Die Erweiterung auf fast doppelt so viele Schalterstufen generiert alternative Pfade im Netz, weshalb Konflikte im Netz durch Umordnen bereits bestehender Verbindungen gelöst werden können. Der große Nachteil gegenüber den nicht-blockierungsfreien Netzen besteht in dem recht aufwändigen Routingverfahren. Das einzige bisher bekannte Routing-Verfahren für das Beneš-Netz ist das *Looping-Routing*. Der Nachteil dieses Verfahrens liegt in dem hohen Anteil an sequentiell auszuführenden Routingschritten, was eine zeitnahe Vermittlung von Verbindungswünschen und damit eine Verwendung für die Übertragung von zeitkritischen Daten ausschließt. Frühere Versuche, das Routing zu dezentralisieren und auf die Schalterebene zu verlagern und dadurch zu beschleunigen, führten zu starken Einschränkungen in der Routingfunktionalität, so dass nicht alle möglichen Permutationen an Verbindungswünschen vermittelt werden konnten. Eine deutliche Verbesserung bietet das *Separation-Routing*, welches ursprünglich im Richter-Netz für zwei hintereinander geschaltete Baseline-Netze entwickelt wurde. Aufgrund einer topologischen Besonderheit des Baseline-Netzes und seiner engen Beziehung zum Beneš-Netz, konnte dieses Verfahren zwar adaptiert werden, allerdings war die Latenz der Leitungsvermittlung in größeren Netzen noch immer viel zu hoch. Eine mögliche Lösung bietet das Routing im Schaltnetz, welches im Rahmen der vorliegenden Arbeit entwickelt wurde. Die Auswertung der binären Zieladressen erfolgt hier mittels kombinatorischer Logik in FPGA-Hardware. Mit dieser Lösung können Verbindungswünsche im Beneš-Netz oder dem Richter-Netz unabhängig von dessen Größe innerhalb einer Gatterlaufzeit und damit weit unterhalb einer Taktperiode vermittelt werden, so dass die Nachricht bereits im nächsten Takt direkt vom Sender an den Empfänger übermittelt werden kann. Nachteilig ist bei dieser Lösung, dass der Hardwareaufwand in der vorgestellten Version nichtlinear mit der Anzahl der Teilnehmer steigt. Stehen jedoch genügend Hardware-Ressourcen zur Verfügung, so bieten die blockierungsfreien Netze eine geeignete Technologie für die Interprozessorkommunikation im Echtzeitparallelrechner *ConPar*.

Kapitel 6 - Energieeffiziente Multiprozessorsysteme

Inhalt. Die Energieeffizienz von Rechensystemen ist generell ein wichtiger Aspekt im Umgang mit endlichen Ressourcen. Bei mobilen Systemen ohne Anbindung an eine permanente Energieversorgung ist die Effizienz sogar von besonderer Bedeutung. Im Falle automobiler Steuergeräte geht der Energieverbrauch zu Lasten der Reichweite des Fahrzeuges. Dieses Kapitel soll verschiedene Möglichkeiten aufzeigen, wie durch *Space-Sharing* und *Software-First-Design* der Energieverbrauch eines Rechensystems flexibel an den Rechenbedarf angepasst und optimiert werden kann.

Ziele und Anforderungen. Bei mobilen Systemen besitzt der effiziente Umgang mit Energie eine besonders herausragende Bedeutung. Während fest installierte Systeme dauerhaft an ein Stromnetz angeschlossen und dadurch kontinuierlich mit Energie versorgt werden können, ist bei mobilen Systemen die zur Verfügung stehende Energiemenge durch die Kapazität des Energiespeichers begrenzt. Beim Automobil bestimmt der Leistungsbedarf des Systems primär die Reichweite, also die Fahrstrecke, die mit einer Tankfüllung oder einer Batterieladung zurückgelegt werden kann. Das Ziel besteht nun darin, den Energiebedarf eines Rechensystems unter Berücksichtigung der benötigten Rechenleistung zu minimieren.

6.1 Grundlagen

Inhalt. Dieser Abschnitt gibt einen Überblick über die verschiedenen Verlustleistungsarten in integrierten Halbleiterbausteinen und zeigt deren Ursachen. Neben der Bedeutung der Energieeffizienz von Rechensystemen werden mögliche Parameter diskutiert, mit denen die Energieeffizienz verbessert werden kann.

Energiekosten. Der Energiebedarf eines Systems setzt sich zusammen aus der Summe aller aufgenommenen Leistungen. Dabei muss generell zwischen zwei Leistungsarten unterschieden werden. Auf der einen Seite steht die Wirkleistung, bei der die zugeführte elektrische Energie in eine Form der Arbeit umgesetzt wird. Auf der anderen Seite steht die Verlustleistung, bei der die zugeführte Energie durch unerwünschte Effekte, z.B. in Form von Wärme, „verloren“ geht.

$$P_{ges} = \sum_i P_i = P_{wirk} + P_{verlust} \qquad (6.1)$$

Wirkleistung vs. Verlustleistung. Da Prozessoren wie auch alle anderen Halbleiterbausteine keine nennenswerten Anteile der zugeführten Energie speichern oder in eine andere Form als thermische Energie wandeln können, strebt der Anteil der Wirkleistung in Rechensystemen gegen Null. Dabei spielt es auch keine Rolle, ob es sich um diskrete Prozessoren oder um Soft-Core Prozessoren handelt. Bei Prozessoren ist die aufgenommene Leistung identisch mit der thermischen Verlustleistung.

$$P_{ges} = \sum P_{verlust} \tag{6.2}$$

Steigerung der Zuverlässigkeit. In integrierten Schaltungen können durch chemische Reaktionen im Halbleitermaterial langfristig irreversible Veränderungen auftreten. Infolgedessen kann es ab einem bestimmten Zeitpunkt dauerhaft zu einem unerwünschten Verhalten des Halbleitermaterials und damit zum Ausfall des Bauelementes kommen. Die Reaktionsgeschwindigkeit solcher Prozesse steigt nach der vom schwedischen Chemiker Arrhenius[1] gefundenen Beziehung exponentiell mit der absoluten Temperatur an. Für integrierte Schaltungen bedeutet dies eine direkte Abhängigkeit der Ausfallrate des Bauteils von der Temperatur des Halbleitermaterials. Die quantitative Beziehung zwischen Temperatur T und Ausfallrate λ wird mathematisch durch die sogenannte Arrhenius-Gleichung 6.3 beschrieben [Sa93, S. 44 f.].

$$\lambda = B \cdot e^{-\frac{E}{kT}} \tag{6.3}$$

Die Ausfallrate wird neben der veränderlichen Temperatur *T* bestimmt durch die materialabhängige Aktivierungsenergie *E*, einem Proportionalitätsfaktor *B* und der Boltzmann-Konstante $k = 8{,}617 \cdot 10^{-5}\ eV/K$. Die in der Gleichung 6.3 enthaltenen Konstanten *E* und *B* sind stark materialabhängig und daher experimentell zu ermitteln. Die **Abbildung 6.1** zeigt in einem Beispiel die Abhängigkeit der Ausfallrate integrierter Schaltungen von der Temperatur des Halbleitermaterials. In der dargestellten Form führt eine dauerhafte Absenkung der Temperatur im Halbleitermaterial, z.B. von 100 °C auf 70 °C, zu einer Reduzierung der Ausfallrate des Bauteils auf ein Drittel und damit zu einer deutlichen Verbesserung der Zuverlässigkeit des Systems. Da es sich, wie bereits ausgeführt, bei der vom Rechensystem aufgenommenen Leistung um thermische Verluste handelt, wird eine Absenkung

[1] Svante August Arrhenius (1859-1927), Nobelpreisträger für Chemie (1903)

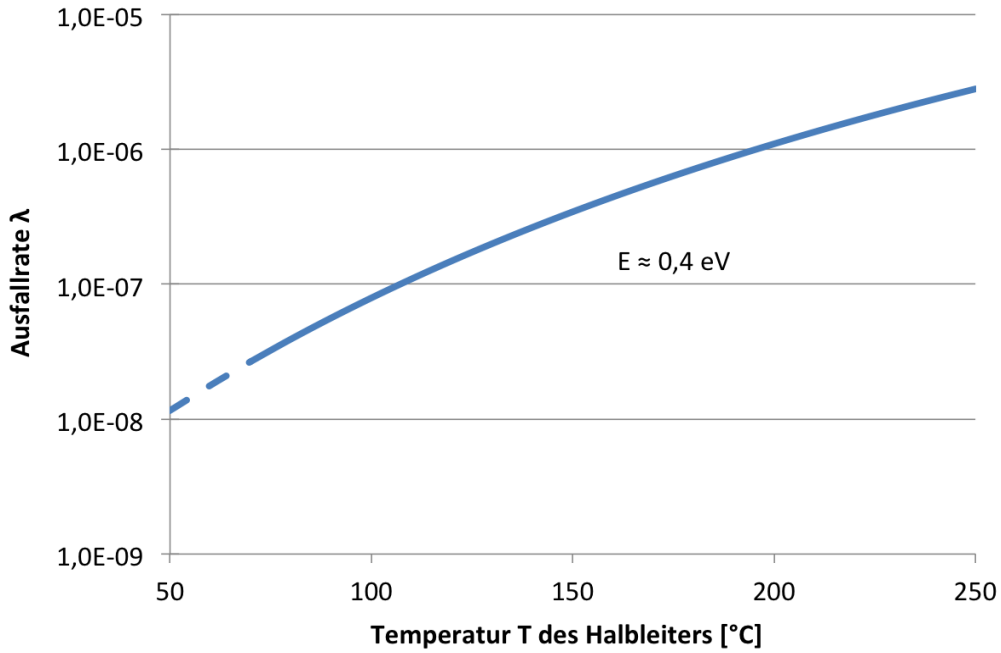


Abbildung 6.1: Ausfallrate von Halbleiter-ICs in Abhängigkeit von der Temperatur

der Verlustleistung durch geeignete Maßnahmen neben den geringeren Energiekosten parallel auch zu einer Steigerung der Zuverlässigkeit des Systems führen.

Aktive Kühlung. Ein weiterer wichtiger Punkt bei den Energiekosten besteht in der aktiven Kühlung des Systems. Meist in der Form eines Lüfters auf dem Chip oder dem Gehäuse wird zusätzliche Energie dafür aufgewendet, die Temperatur der Halbleiter aktiv zu senken. In großen Rechenzentren werden sogar bis zu 50 Prozent des gesamten Energiebedarfs zur Kühlung der Prozessoren verwendet. So benötigt das Münchner Leibniz-Rechenzentrum für seinen im Jahr 2006 in Betrieb genommenen Neubau eine Leistung von zwei Megawatt allein für die Kühlung der Rechner [Fi10, S. 2]. Durch die Senkung der Verlustleistung im Rechensystem bzw. im Steuergerät könnte der zusätzliche Strombedarf für die aktive Kühlung reduziert oder sogar ganz eliminiert werden.

Steigerung der Energieeffizienz. Natürlich kann man darüber streiten, ob bei einem rein verlustbehafteten System eine Steigerung der Energieeffizienz überhaupt möglich ist. Im Normalfall ist es jedoch so, dass Rechensysteme, z.B. in Form eines Steuergerätes, Teil eines übergeordneten Systems sind. Und dessen Energiebilanz hängt sehr wohl auch von der Verlustleistung der Steuerung ab. Unabhängig davon bleibt die thermische Verlustleistung der entscheidende Parameter bei der Analyse des Energieverbrauchs von Steuergeräten. Die Verlustleistung kann im FPGA,

wie auch bei anderen elektronischen Halbleitern, je nach Ursache der Verluste in die statische Verlustleistung und die dynamische Verlustleistung unterschieden werden [Ye98]. Beide Arten sollen in den nachfolgenden Abschnitten näher betrachtet werden.

$$P_{verlust} = \sum P_{stat} + \sum P_{dyn} \qquad (6.4)$$

6.1.1 Statische Verlustleistung

Ursache und Wirkung. In integrierten Schaltungen bestehen die Ursachen für statische Verlustleistung hauptsächlich in Leck- und Bias-Strömen an den p-n-Übergängen der Transistoren, welche durch Diffusion von Elektronen bzw. Löchern zwischen den unterschiedlich dotierten Halbleiterschichten auftreten. Je feiner die geometrischen Abmessungen der Halbleiter und je dünner die Oxidschichten ausgeführt sind, desto größer sind die im Halbleitermaterial auftretenden statischen Verluste[2]. Dieser Zusammenhang ist vor allem in Hinblick auf die weiter voranschreitende Miniaturisierung integrierter Schaltungen von Bedeutung. Die **Abbildung 6.2** zeigt am Beispiel eines MOS-Transistors zwei typische Verlustströme, wie sie auch in Prozessoren vorkommen. Ein idealer p-n-Übergang würde den Stromfluss sperren, sobald eine positive Spannung an die n-dotierte Schicht (Sperrspannung, engl. *reverse bias*) gelegt wird. In der Praxis fließt jedoch ein geringer Sperrstrom $I_{reverse}$, der dem Durchlassstrom entgegen gerichtet ist. Feinere Strukturen bedeuten in diesem Fall, dass die Sperrschicht durch die verkürzte Länge des Gates dünner wird und somit den unerwünschten Sperrstrom weiter begünstigt. Die zweite Leckage $I_{subthreshold}$ tritt zwar in positiver

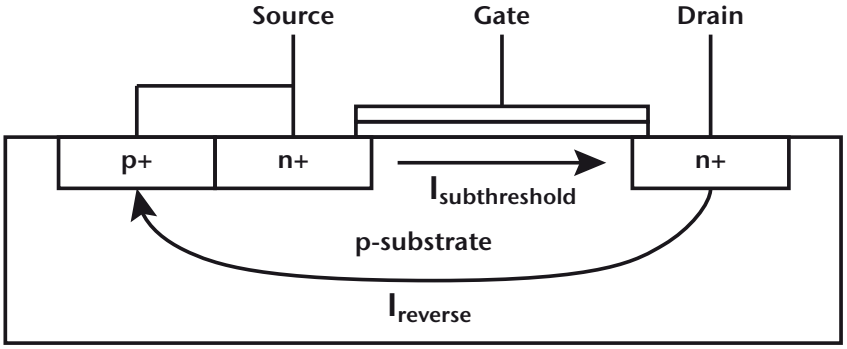


Abbildung 6.2: Leckage-Ströme in einem CMOS-Gate[3]

[2] vgl. [Ye98, S. 15 ff.]

[3] grafische Darstellung aus [Ye98, S. 16]

Richtung auf, erscheint aber bereits im ausgeschalteten Zustand, während sich die angelegte Spannung noch unterhalb der Schwellspannung $U_{threshold}$ befindet, ab welcher der p-n-Übergang leitend wird. Da mit der abnehmenden Strukturgröße auch die erforderliche Betriebsspannung sinkt, sind gleichbleibende Taktgeschwindigkeiten nur durch reduzierte Schwellwertspannungen möglich. Dadurch erhöht sich allerdings der Diffusionsstrom unterhalb der Schwellwertspannung. Wie die (Gleichung 6.5) zeigt, ergibt sich die statische Verlustleistung P_{stat} der Schaltung aus der angelegten Betriebsspannung U_B und der Summe aller auftretenden Leck- und Bias-Ströme.

$$P_{stat} = U_B \cdot \sum I_{leckage} \qquad (6.5)$$

Bedeutung. Die statische Verlustleistung erfährt vor allem beim VLSI-Design integrierter Schaltungen eine wachsende Bedeutung[4]. Mit der fortlaufenden Miniaturisierung der Strukturen wird der Anteil der statischen Verlustleistung an der Gesamtverlustleistung in der nahen Zukunft, wie auch in der Vergangenheit, deutlich zunehmen (**Abbildung 6.3**).

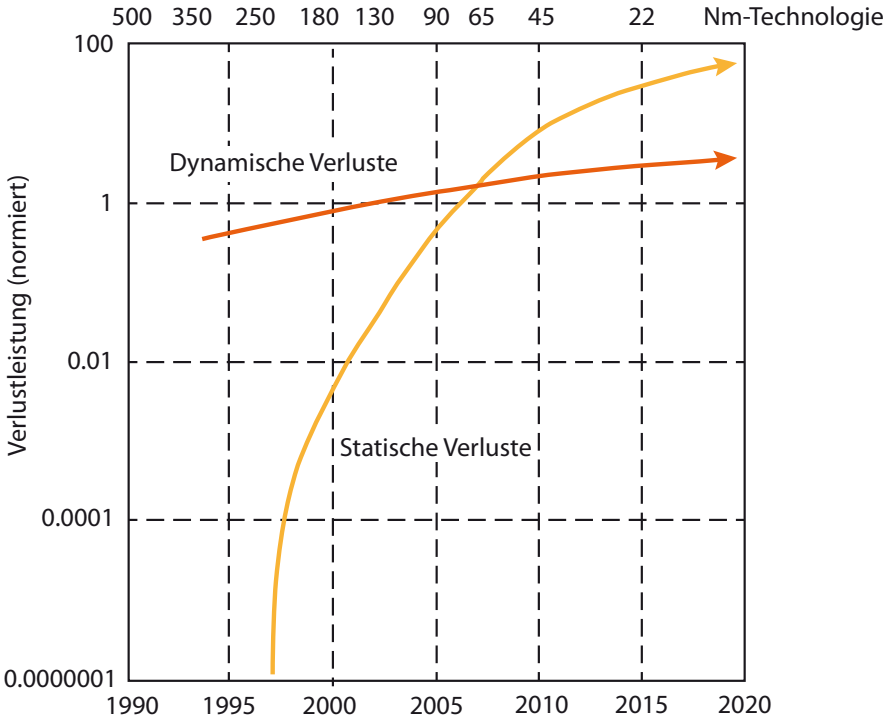


Abbildung 6.3: Statische und dynamische Verlustleistung vs. Nm-Technologie[5]

[4] vgl. [Ki03]

[5] grafische Darstellung aus [Te05, S. 3]

Optionen zur Optimierung. Da die statische Verlustleistung primär durch das Design des Halbleiterherstellers der integrierten Schaltungen beeinflusst wird, sind die Möglichkeiten zur Einflussnahme während der Laufzeit sehr eingeschränkt. Im Falle von Space-Sharing könnte eine Möglichkeit zum Beispiel darin bestehen, die Stromversorgung nicht benötigter Partitionen im Multiprozessorsystem temporär abzuschalten. Allerdings muss tatsächlich eine galvanische Trennung des Halbleiters von der Stromversorgung erfolgen, da die statische Verlustleistung taktunabhängig ist. Eine einfache Taktabschaltung z.B. durch Gated Clocks ist in diesem Fall nicht ausreichend. Dazu wären jedoch grundlegende Änderungen am FPGA-Design notwendig, die zum Zeitpunkt und im Rahmen dieser Arbeit noch nicht abzusehen sind. Aus diesem Grund soll die statische Verlustleistung trotz ihrer steigenden Bedeutung an der Gesamtverlustleistung an dieser Stelle nicht weiter betrachtet werden.

6.1.2 Dynamische Verlustleistung

Ursache und Wirkung. Die dynamische Verlustleistung entsteht durch Schaltaktivitäten der Halbleiterelemente auf dem Chip. Die beim Schaltvorgang auftretenden Verluste können je nach Ursache in zwei Kategorien unterteilt werden. Die erste Kategorie sind Querströme. Da eingehende Spannungen in der Praxis keine unendlichen Flankensteilheiten aufweisen, können im Moment des Umschaltens kurze Stromspitzen auftreten, während der p-Kanal und der n-Kanal gleichzeitig leitend sind. Die zweite Kategorie sind Umladeströme, die durch Auf- bzw. Entladungen von Knoten- und Verbindungskapazitäten auf dem Chip entstehen[6]. Diese kapazitiven Verluste lassen sich wie folgt beschreiben[7]:

$$P_{dyn} = \sum_i C_i \cdot U_i^2 \cdot f_i \qquad (6.6)$$

Wie in der Gleichung 6.6 angegeben, wird die dynamische Verlustleistung P_{dyn} von digitalen Schaltungen bestimmt durch die Summe aus der Schaltkapazität C, der Schaltspannung U und der Schaltfrequenz f der einzelnen Komponenten, wobei die Spannung quadratisch in das Ergebnis eingeht, während die Frequenz ebenso wie die Kapazität lediglich einen linarenn Einfluss besitzt.

[6] vgl. [Fi10, S.11 ff.]

[7] aus [Sh02, S. 158]

Bedeutung. Insbesondere bei Prozessoren finden relativ viele Umschaltvorgänge in sehr kurzen Zeitintervallen statt. Das bedeutet, dass der Anteil der dynamischen Verlustleistung P_{dyn} an der Gesamtverlustleistung in einem Multiprozessorsystem relativ hoch ausfällt. Gleichzeitig bietet sich (im Gegensatz zur statischen Verlustleistung) die Möglichkeit, während der Laufzeit mit relativ einfachen Mitteln Einfluss auf die Verlustleistung zu nehmen.

Optionen zur Optimierung. Um die dynamische Verlustleistung P_{dyn} zu reduzieren, bieten sich gemäß der Gleichung 6.6 die folgenden drei Optionen an:

- Senkung der Schaltkapazitäten
- Senkung der Schaltspannung
- Senkung der Schaltfrequenz

Die genannten Möglichkeiten zur Reduzierung der dynamischen Verlustleistung sollen in den folgenden Abschnitten kurz diskutiert werden.

Senkung der Schaltkapazitäten. Die Größe der Schaltkapazitäten hängt ganz wesentlich vom Design der Halbleiterstrukturen ab. In einem FPGA kann die Verlustleistung z.B. im Place&Route-Prozess durch ein geeignetes Layout verringert werden. Platzierungswerkzeuge reduzieren die Länge der Verbindungen im FPGA durch geeignete Platzierung der einzelnen Komponenten. Router haben die Möglichkeit, die Wegewahl der Verbindungen zu optimieren und so die Schaltkapazitäten zu minimieren. Untersuchungen der Firma Xilinx haben gezeigt, dass die dynamische Verlustleistung durch optimierte Layouts bei Spartan-3-FPGAs bis zu 14%, bei Virtex-4-FPGAs 11% und für Virtex-5-FPGAs 12% reduziert werden konnte [Co09, S. 37].

Senkung der Schaltspannung. Da die Schaltspannung quadratisch in die dynamische Verlustleistung eingeht (Gleichung 6.6), besitzt die Absenkung der Schaltspannung das größte Einsparpotential aller drei Optionen. Allerdings kann dieser Schritt nur in Verbindung mit einer Absenkung der Taktfrequenz erfolgen, da die Schaltgeschwindigkeit durch die verringerte Schaltspannung absinkt. Führende Hersteller nutzen diese Kombination bereits, um die Leistungsaufnahme bei diskreten Prozessoren zu verringern. Während bei mobilen Anwendungen wie Notebooks primär die Verlängerung der Laufzeit eine Rolle spielt, stehen bei Desktop- und Server-Anwendungen Hitze- bzw. Kühlprobleme im Vordergrund.

Die Firma Intel hat deshalb die „Enhanced Intel SpeedStep Technologie“ (EIST) als stromsparende Maßnahme entwickelt und für verschiedene Prozessoren in Notebooks, aber auch in Desktop- und Serveranwendungen eingeführt. Mit Hilfe der EIST ändert der Prozessor seine Taktrate, also die Rechenleistung, je nach Einstellung und Bedarf. Bei reduzierter Taktrate lässt sich die Kernspannung des Prozessors ebenfalls senken. Die Firma AMD hat für ihre Prozessoren ebenfalls Techniken zur Verringerung der Verlustleistung entwickelt. Diese Technologien sind unter den Namen „PowerNow!“ für Notebooks und „Cool’n’Quiet“ für Arbeitsplatzrechner und Server verfügbar. Auch hier werden Kernspannung und Taktrate des Prozessors an die aktuellen Anforderungen der Rechenleistung angepasst.

Senkung der Schaltfrequenz. Die Senkung der Schalt- bzw. Taktfrequenz ist ebenfalls ein probates Mittel, welches von Prozessorherstellern in Verbindung mit der Absenkung der Schaltspannung eingesetzt wird, um die dynamische Verlustleistung zu senken. Dabei wird die Taktfrequenz des Prozessors so weit abgesenkt, wie es die aktuell benötigte Rechenleistung und die minimale Schaltspannung (*threshold voltage*) erlauben. Es gilt[8]:

$$\frac{f_2}{f_1} \approx \sqrt[3]{\frac{P_2}{P_1}} \qquad (6.7)$$

Ohne die Absenkung der Schaltspannung verhält sich die dynamische Verlustleistung lediglich proportional zur Taktfrequenz (vgl. Gleichung 6.6). Während die Absenkung der Schaltkapazitäten und -spannungen von Prozessoren und FPGAs aufgrund der aufwändigen Realisierung den Herstellern vorbehalten bleibt, kann die Anpassung der Taktfrequenz mit relativ einfachen Mitteln umgesetzt werden. Insbesondere für die FPGA-basierten Softprozessoren bietet sich diese Möglichkeit durch die Verwendung von Taktgeneratoren bzw. Taktteilern auf dem Chip an.

Ausnutzen von Zeitschranken. Das Rechnen in einem Echtzeitsystem ist von Zeitschranken geprägt, innerhalb derer die einzelnen Aufgaben erfüllt werden müssen. Das Ziel ist nicht, eine Aufgabe so schnell wie möglich zu erledigen, sondern lediglich so schnell wie nötig, also vor dem Erreichen der jeweiligen Zeitschranke (Deadline). Die **Abbildung 6.4** zeigt an einem Zahlenbeispiel, wie der Energieverbrauch des Rechensystems verbessert werden kann, wenn für die Erledigung einer Aufgabe von 1000 Millionen Zyklen die gesamte vorhandene

[8] vgl. [Fl99, S. 18]

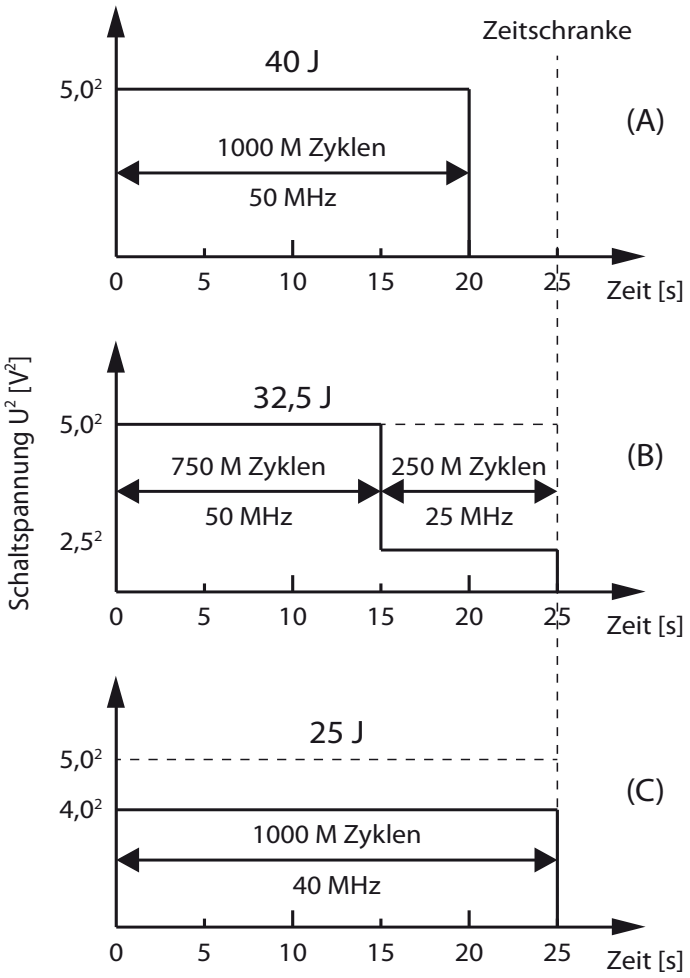


Abbildung 6.4: Energieverbrauch in Abhängigkeit von Schaltfrequenz und -spannung[9]

Zeitspanne von 25 Sekunden ausgenutzt wird. Ausgehend von einer maximalen Rechenleistung (A) werden Schaltspannung und Schaltfrequenz teilweise (B) oder vollständig (C) reduziert, soweit wie es die Zeitschranke erlaubt. Durch das Ausnutzen der Zeitschranke könnte der Energieverbrauch des Rechensystems in dem Beispiel von 40 Joule (A) auf bis zu 25 Joule (C) gesenkt werden.

6.2 Verlustleistungen im FPGA-basierten Multiprozessorsystem

Inhalt. In diesem Abschnitt soll geklärt werden, aus welchen Komponenten ein FPGA-basiertes Multiprozessorsystem besteht und welchen Anteil die einzelnen Komponenten an der dynamischen Verlustleistung des Systems besitzen. Zur Klärung dieser Frage wurden Leistungsmessungen mit unterschiedlichen Multiprozessorsystemen an verschiedenen FPGAs durchgeführt. Die Ergebnisse dieser

[9] grafische Darstellung aus [Ya06, S. 186]

Messungen werden kurz vorgestellt und diskutiert. Die Messungen selbst sind im Anhang B dieser Arbeit ausführlich dokumentiert.

Aufbau eines FPGA-basierten Multiprozessorsystems. Damit die Messungen erfolgreich durchgeführt werden können, muss zunächst geklärt werden, aus welchen Komponenten ein FPGA-basiertes Multiprozessorsystem besteht und welche Parameter untersucht werden sollen. Grundsätzlich setzt sich ein Multiprozessorsystem aus drei Komponenten zusammen:

- Prozessoren
- Speicher
- Verbindungsnetzwerk

Für die Untersuchung der dynamischen Verlustleistung P_{dyn} kommen die Taktfrequenz und die Größe des Systems als vom Anwender veränderliche Parameter in die nachfolgenden Betrachtungen.

Messobjekte. Die Messungen wurden an handelsüblichen Entwicklungsboards mit FPGAs der Firma Xilinx durchgeführt[10]. Insgesamt standen für die Messungen drei Xilinx FPGAs der Familien Spartan-3, Virtex-4 und Virtex-5 zur Verfügung. Als Softprozessor diente der proprietäre MicroBlaze, ein 32-Bit RISC-Prozessor von Xilinx [Xi08b].

6.2.1 Einfluss der Taktfrequenz

Einfluss der Taktfrequenz. Zunächst soll die Frage geklärt werden, ob die Frequenz des vom FPGA-internen Taktgenerator erzeugten Systemtaktes einen Einfluss auf die Stromaufnahme und damit auf die Verlustleistung des FPGAs hat. Hierzu wurde eine Multiprozessorarchitektur implementiert, welche mit Hilfe eines Taktteilers von extern über DIP-Schalter mit unterschiedlichen Frequenzen getaktet werden kann. Da Soft-Prozessoren bis auf Null herunter getaktet werden können, lässt sich der statische Anteil der Verlustleistung relativ einfach durch Anhalten des Prozessortaktes herausfiltern. Die **Abbildung 6.5** zeigt das Ergebnis von Messungen, die an Xilinx FPGAs vom Typ Spartan-3 und Virtex-5 vorgenommen wurden[11]. In der Abbildung 6.5 ist der lineare Einfluss der Taktfrequenz f_{takt} auf die dynamische

[10] vgl. Tabelle B.1 : Liste der untersuchten FPGAs, S. 278

[11] vgl. B.2 Verlustleistung in Abhängigkeit von der Taktfrequenz, S. 280 ff.

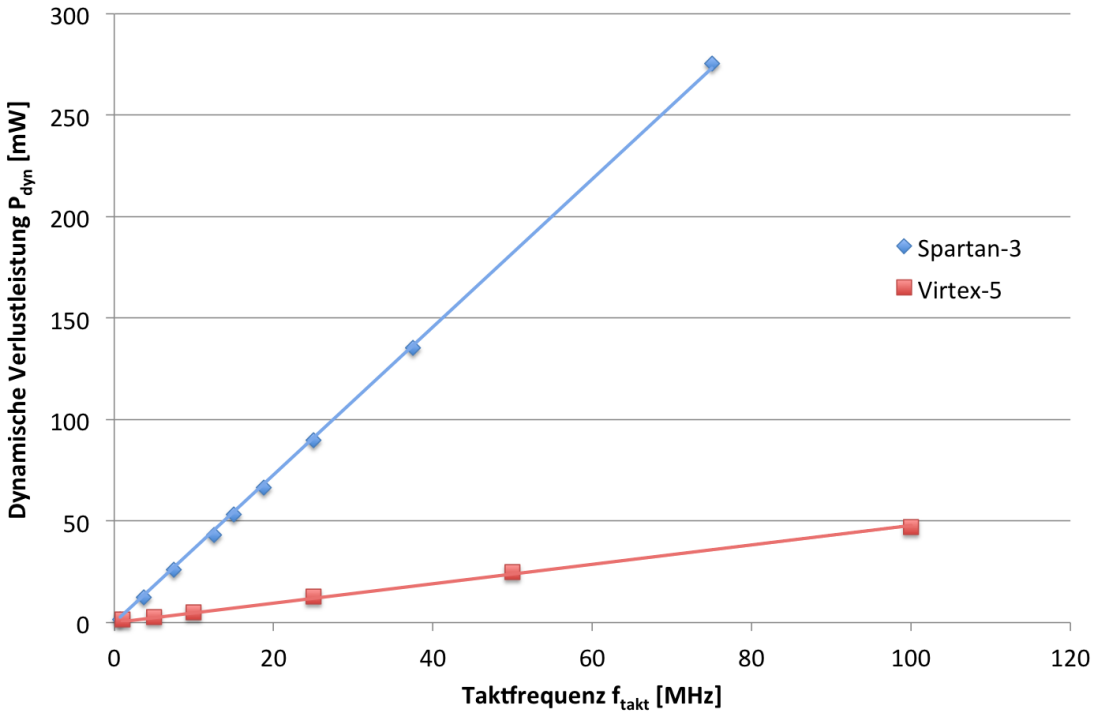


Abbildung 6.5: Dynamische Verlustleistung eines einzelnen Softprozessors in Abhängigkeit von der Taktfrequenz am Beispiel von Spartan-3 und Virtex-5 FPGAs

Verlustleistung sehr gut zu erkennen. Aus der Steigung der beiden Geraden ergibt sich eine frequenzabhängige dynamische Verlustleistung $P_{dyn}(f_{takt}) \approx 3{,}64\,mW/MHz$ beim Spartan-3 FPGA bzw. $P_{dyn}(f_{takt}) \approx 0{,}48\,mW/MHz$ beim Virtex-5 FPGA. Der Zusammenhang aus Gleichung 6.6, S. 174 ist somit bestätigt worden.

6.2.2 Einfluss der Prozessorzahl

Einfluss der Anzahl der Softprozessoren. Nach dem Nachweis über die Abhängigkeit der dynamischen Verlustleistung P_{dyn} des Prozessors von der Taktfrequenz f_{takt} soll nun untersucht werden, ob und wie die vom FPGA aufgenommene Verlustleistung von der Anzahl der Prozessoren im MPSoC abhängt. Um diese Frage zu beantworten, werden weitere Messungen an Xilinx FPGAs vom Typ Spartan-3, Virtex-4 und Virtex-5 vorgenommen[12]. Hierzu werden Multiprozessorsysteme mit einer unterschiedlichen Anzahl an Softprozessoren in den Spartan-3 und Virtex-5 FPGAs implementiert und die Leistungsaufnahme bei verschiedenen Taktfrequenzen gemessen. Das Ergebnis der Messungen ist in der **Abbildung 6.6** (für Spartan-3) und in der **Abbildung 6.7** (für Virtex-5) grafisch dargestellt.

[12] vgl. B.3 Verlustleistung in Abhängigkeit von der Prozessoranzahl, S. 286 ff.

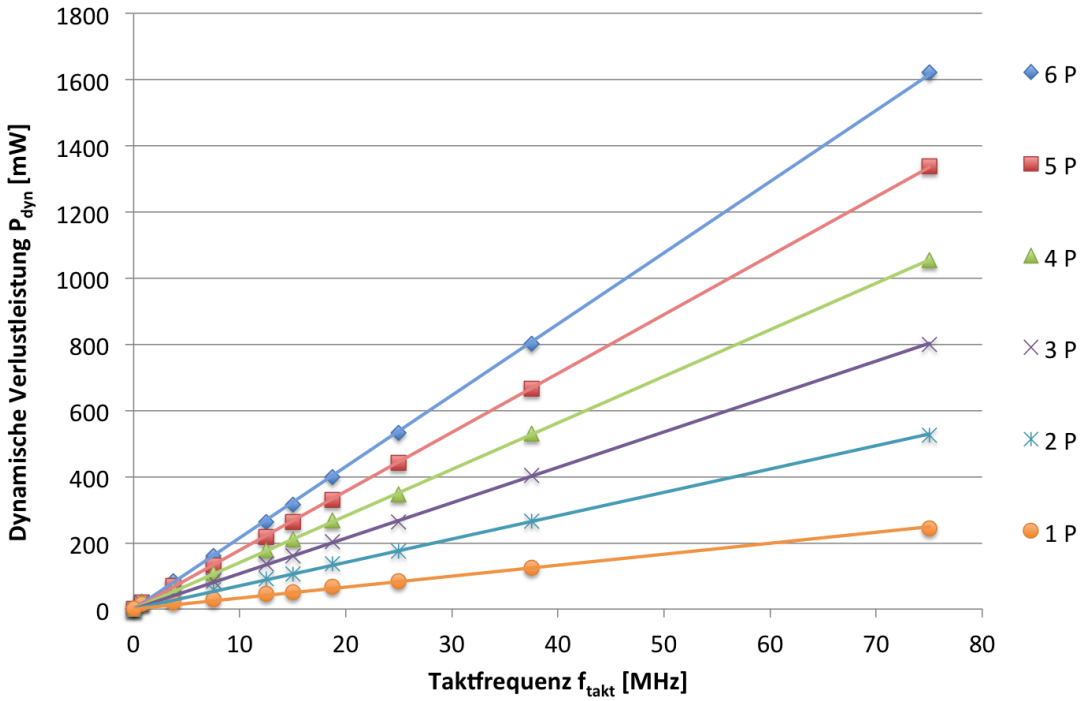


Abbildung 6.6: Dynamische Verlustleistung von Multiprozessorsystemen unterschiedlicher Größen in Abhängigkeit von der Taktfrequenz am Beispiel des Spartan-3 FPGA

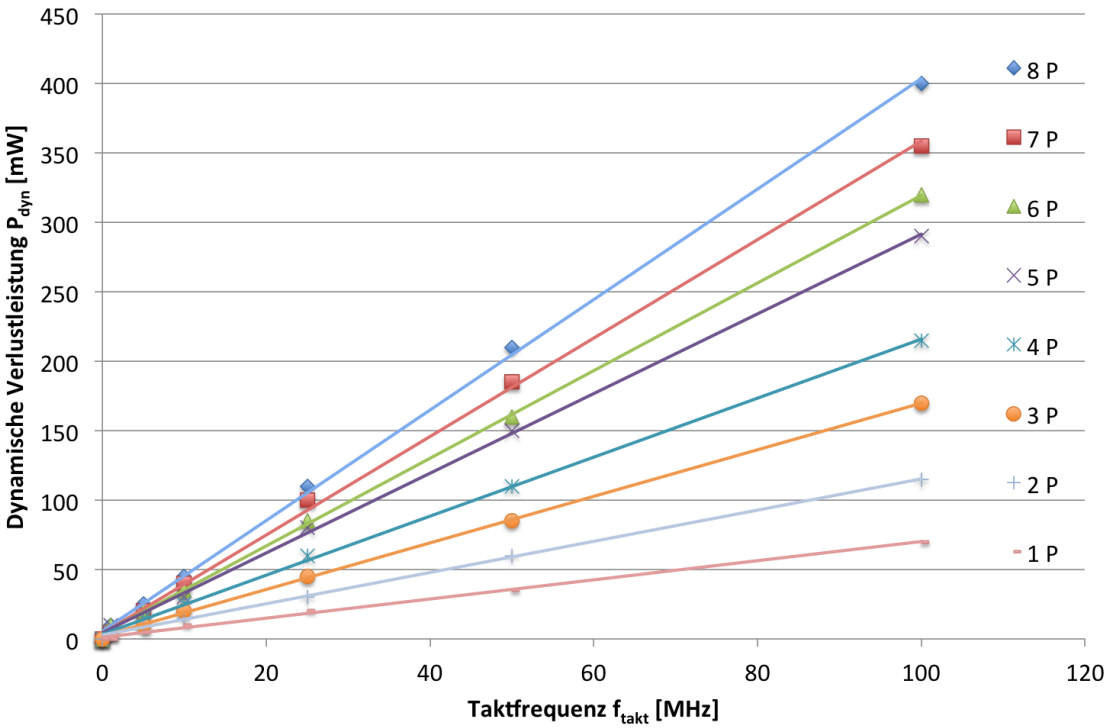


Abbildung 6.7: Dynamische Verlustleistung von Multiprozessorsystemen unterschiedlicher Größen in Abhängigkeit von der Taktfrequenz am Beispiel des Virtex-5 FPGA

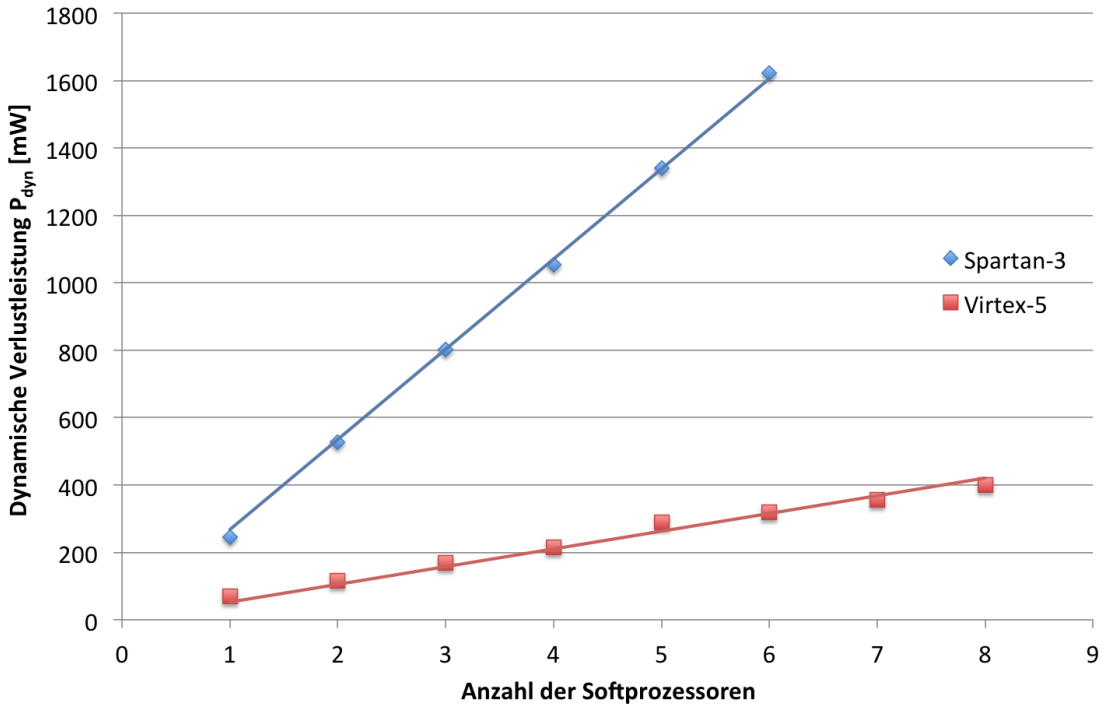


Abbildung 6.8: Abhängigkeit der dyn. Verlustleistung von der Anzahl der Softprozessoren im Multiprozessorsystem am Beispiel von Spartan-3 und Virtex-5 FPGAs

Sofort auffällig ist bei allen untersuchten Multiprozessorsystemen der bereits aus der Abbildung 6.5 bekannte lineare Zusammenhang zwischen der dynamischen Verlustleistung P_{dyn} und der Taktfrequenz f_{takt}. In den beiden Abbildungen ist zu erkennen, dass die Steigung der Geraden mit der Anzahl der Prozessoren zunimmt, wobei der Abstand zwischen den Geraden relativ konstant bleibt. Das bedeutet, dass ein direkter Zusammenhang zwischen der dynamischen Verlustleistung P_{dyn} und der Größe des Multiprozessorsystems existiert. Abweichungen, die sich vor allem beim Virtex-5 FPGA bemerkbar machen, lassen sich durch die unterschiedlichen Platzierungen und Synthesen der einzelnen Komponenten im FPGA erklären.

Die **Abbildung 6.8** zeigt den Zusammenhang zwischen der dynamischen Verlustleistung P_{dyn} und der Anzahl der Softprozessoren aus den gemittelten Werten von Abbildung 6.6 und Abbildung 6.7. Aus der Steigung der Regressionsgeraden ergibt sich für den Spartan-3 FPGA eine dynamische Verlustleistung $P_{dyn} \approx 268\ mW$ für jeden Softprozessor bei einer Taktfrequenz $f_{takt} = 75\ MHz$. Für den Virtex-5 FPGA ergibt sich eine dynamische Verlustleistung $P_{dyn} \approx 53$ mW für jeden Softprozessor bei einer Taktfrequenz $f_{takt} = 100\ MHz$. Im direkten Vergleich der Verläufe in Abbildung 6.8 wird sichtbar, dass die dynamische Verlustleistung P_{dyn} beim

Virtex-5 deutlich niedrigerer ausfällt als beim Spartan-3. Die Erklärung hierfür liegt in der technischen Weiterentwicklung des Virtex-5 gegenüber dem Spartan-3 FPGA. So wurde der Virtex-5 bereits in der 65-nm-Technologie gefertigt, während der Spartan-3 noch in der 90-nm-Technologie ausgeführt ist. Dadurch konnte zum Beispiel die Kernspannung von 1,2 V auf 1,0 V gesenkt werden.

Mehr Softprozessoren im Virtex-4 FPGA. Um ein noch genaueres Bild von der Abhängigkeit der dynamischen Verlustleistung P_{dyn} von der Anzahl der Softprozessoren auf dem FPGA zu bekommen, wurde eine zusätzliche Messreihe am deutlich größeren Virtex-4 FPGA XC4VFX100 aufgenommen. In diesem FPGA konnten für die Messungen bis zu 34 Softprozessoren vom Typ Xilinx MicroBlaze implementiert werden. Zum Vergleich: der Spartan-3 FPGA XC3S1000 (Abbildung 6.8) bietet Platz für maximal 6 Softprozessoren. Die **Abbildung 6.9** zeigt das Ergebnis der Messungen. Aus der Steigung der Regressionsgeraden ergibt sich für den Virtex-4 FPGA eine dynamische Verlustleistung $P_{dyn} \approx 120\ mW$ für jeden Softprozessor bei einer Taktfrequenz $f_{takt} = 100\ MHz$. Auch hier sind im Verlauf deutliche Abweichungen einzelner Messpunkte von der Geraden zu erkennen, deren Ursache im Place&Route-Werkzeug sowie der Schaltungssynthese liegt. Die Abweichungen

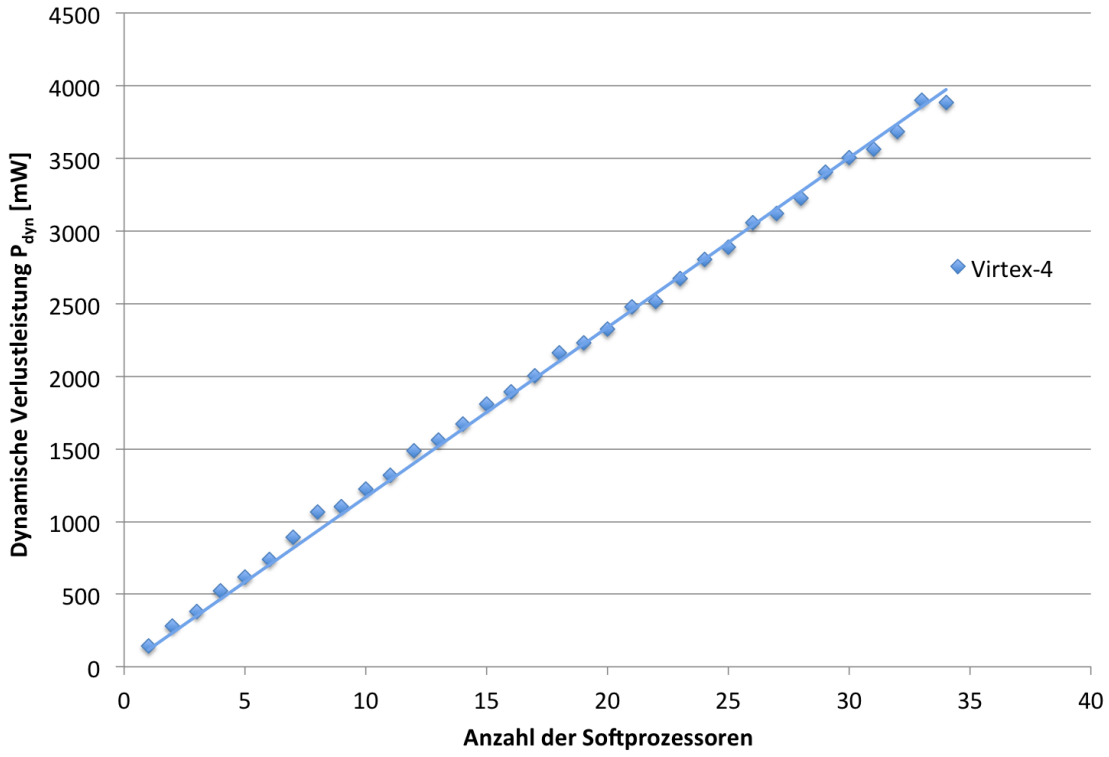


Abbildung 6.9: Abhängigkeit der dynamischen Verlustleistung von der Anzahl der Softprozessoren im Multiprozessorsystem am Beispiel des Virtex-4 FPGA

in absoluten Zahlen bleiben auch bei steigender Zahl der Softprozessoren konstant, so dass die maximale Abweichung der Messpunkte von der eingezeichneten Geraden bei sehr großen Multiprozessorsystemen mit hundert oder mehr Prozessoren extrapoliert eher im Zehntel-Prozent-Bereich liegen dürfte.

6.2.3 Statische vs. dynamische Verlustleistung

Verhältnis von statischer zu dynamischer Verlustleistung. Die **Abbildung 6.10** zeigt am Beispiel der Ergebnisse der Messungen am Virtex-4 FPGA, wie sich die Anteile von statischer und dynamischer Verlustleistung zur Gesamtverlustleistung im FPGA-basierten Multiprozessorsystem zusammensetzen. Ein ähnlicher Verlauf zur Entwicklung von statischer und dynamischer Verlustleistung war bereits in der Abbildung 6.3, S. 173 zu sehen. Allerdings stehen dort die beiden Verlustleistungen nicht in Relation zueinander, so dass zwar der Trend für jede Kurve, nicht aber das anteilige Verhältnis der beiden Verlustleistungsarten zueinander erkennbar ist. Aus der Abbildung 6.10 ist nun zu erkennen, dass mit der Größe des Multiprozessorsystems auch der Anteil der dynamischen Verlustleistung mit etwa

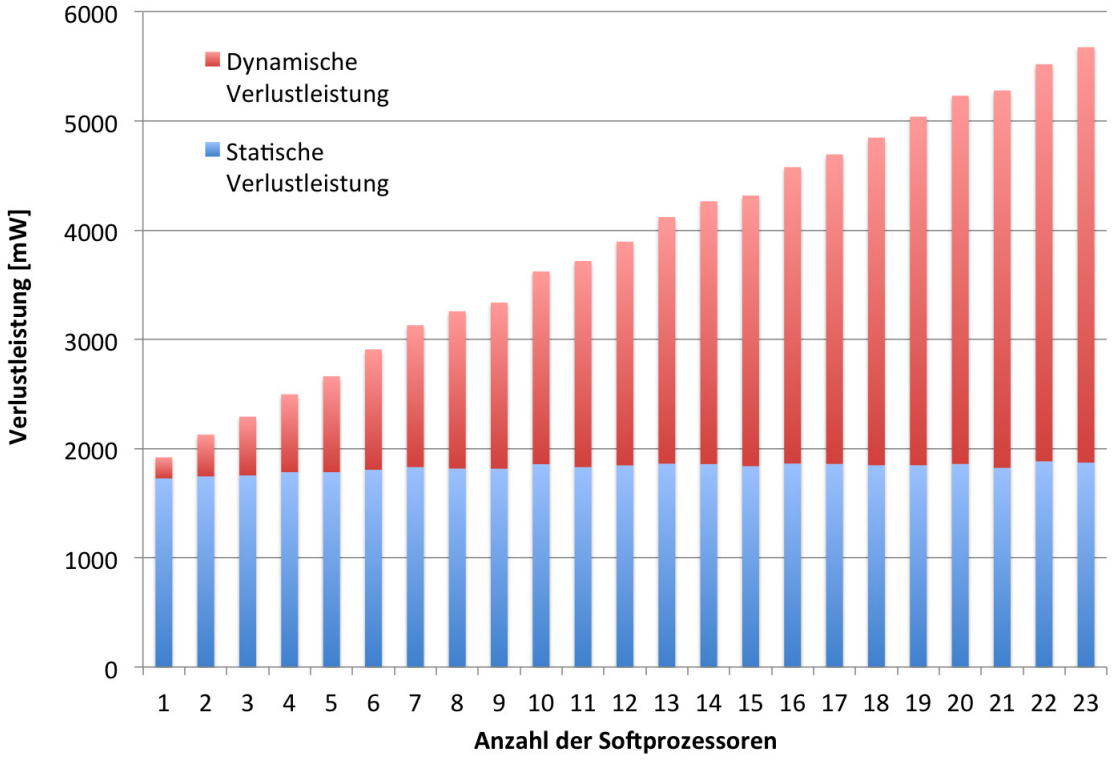


Abbildung 6.10: Abhängigkeit von statischer und dynamischer Verlustleistung von der Anzahl der Softprozessoren am Beispiel des Virtex-4 FPGA

120 mW pro Prozessor steigt (vgl. Abbildung 6.9), während der statische Anteil mit ungefähr 1800 mW über den gesamten Bereich konstant bleibt. In dem dargestellten Beispiel überwiegt der Anteil der dynamischen Verlustleistung ab einer Größe von 12 Prozessoren.

6.2.4 Einfluss der Lokalspeicher

Einfluss der Lokalspeicher. Xilinx FPGAs bieten die Möglichkeit, FPGA-interne Block-RAMs als lokale Prozessorspeicher zu nutzen. Jeder Softprozessor ist mit zwei Speichern ausgestattet, die über zwei getrennte Speicherbusse (LMB) an den Softprozessor gebunden sind. Dadurch stehen zwei getrennte Speicher jeweils für Befehle und für Daten zur Verfügung. Da die Größe der Lokalspeicher je nach FPGA-Typ innerhalb definierter Grenzen variiert werden kann, sollte nun durch Messungen am FPGA die Frage geklärt werden, ob die gewählte Speichergröße einen signifikanten Einfluss auf die dynamische Verlustleistung P_{dyn} des Multiprozessorsystems besitzt. Für die Messungen wurde ein Xilinx Virtex-4 FPGA verwendet, da dieser deutlich mehr Block-RAM als der Spartan-3 oder der Virtex-5 besitzt und daher mehr Messwerte erlaubt. In den Messungen wurden unterschiedlich große Multiprozessorsysteme mit 1 bis 5 Softprozessoren und einer Lokalspeichergröße von 32, 64, 128 und 256 KiB[13,14] untersucht.

Die **Abbildung 6.11** zeigt das Ergebnis der Messungen in der grafischen Übersicht. Gut zu erkennen ist in allen Messreihen der lineare Zusammenhang zwischen der dynamischen Verlustleistung P_{dyn} und der Größe der Lokalspeicher. Die Steigungen der Geraden nehmen mit der Anzahl der Softprozessoren ebenfalls linear zu, das heißt auch die Anzahl der Lokalspeicher ist für die Verlustleistung des Multiprozessorsystems von Bedeutung. Ein weiterer Gesichtspunkt ist der Schnittpunkt der Geraden mit der vertikalen Achse. Dieser Punkt zeigt an, wieviel Leistung ohne Speicher, also nur für die Ausführung der Logik des Softprozessors in Hardware, benötigt wird. Auch hier besteht ein linearer Zusammenhang zwischen der dynamischen Verlustleistung und der Anzahl der Softprozessoren. Aus allen genannten Punkten ergibt sich nun ein relativ genaues Bild über die dynamische Verlustleistung in einem Multiprozessorsystem. Im Falle des für die Messungen verwendeten Virtex-4 FPGA benötigt jeder

[13] 1 KiB = 2^{10} Byte, Definition nach IEC 80000-13:2008

[14] sprich: 1 KibiByte

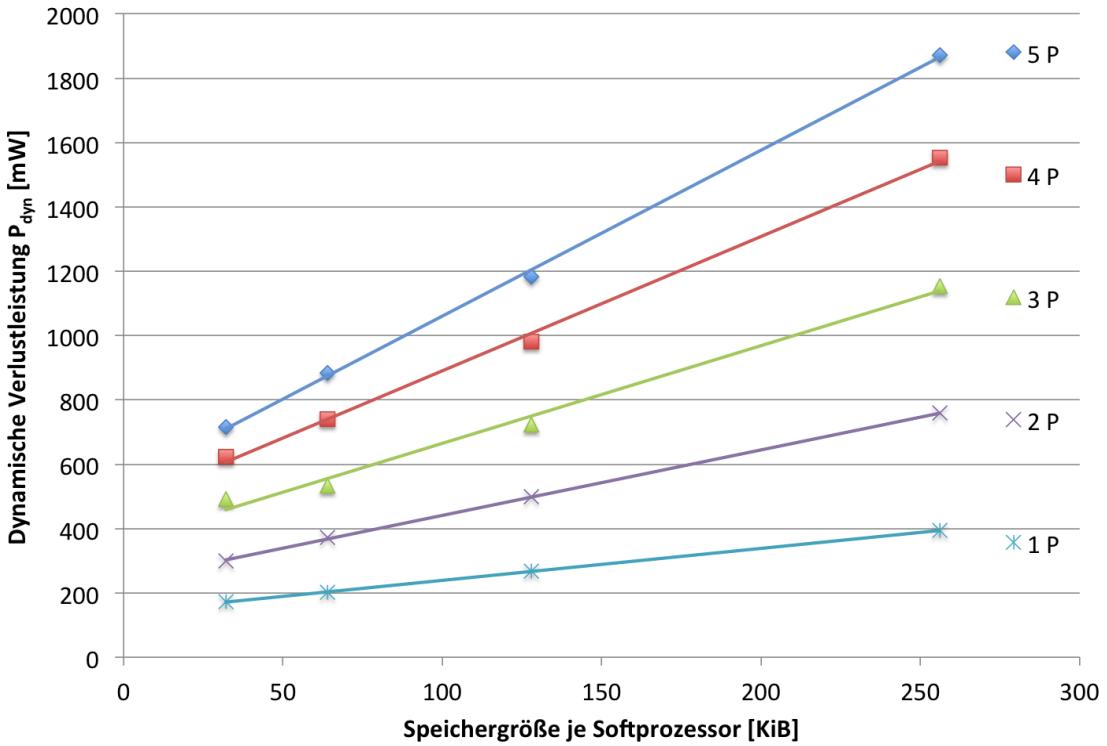


Abbildung 6.11: Abhängigkeit der dynamischen Verlustleistung von der Speichergröße am Beispiel des Virtex-4 FPGA

Softprozessor eine durchschnittliche Verlustleistung von $P_{dyn}(f_{takt}) = 1\ mW/MHz$ zuzüglich einer speicherabhängigen Verlustleistung von $P_{dyn}(f_{takt}) = 0{,}01\ mW/MHz$ pro KiB Speicher[15].

6.2.5 Einfluss des Verbindungsnetzwerks

Einfluss des Verbindungsnetzwerks. Eine weitere Komponente bei der Betrachtung der dynamischen Verlustleistung in einem Multiprozessorsystem stellt das Verbindungsnetzwerk für die Interprozessorkommunikation dar. Wie schon bei den vorhergehenden Betrachtungen der Prozessoren stellt sich auch beim Verbindungsnetzwerk die Frage, ob und wie die Eigenschaften des Verbindungsnetzwerkes die Leistungsaufnahme des Multiprozessorsystems beeinflussen. Die **Abbildung 6.12** zeigt das Ergebnis von Messungen, die an einem Multiprozessorsystem mit 8 Softprozessoren und einem einfachen Baseline-Netz als Verbindungsnetzwerk auf einem Virtex-4 FPGA vorgenommen wurden[16]. Das Ziel dieser Messungen war der Nachweis über die Abhängigkeit der dynamischen Verlustleistung P_{dyn} von der

[15] vgl. B.4 Verlustleistung in Abhängigkeit von der Speichergröße, S. 292 ff.

[16] vgl. B.5 Verlustleistung in Abhängigkeit vom Verbindungsnetzwerk, S. 295 ff.

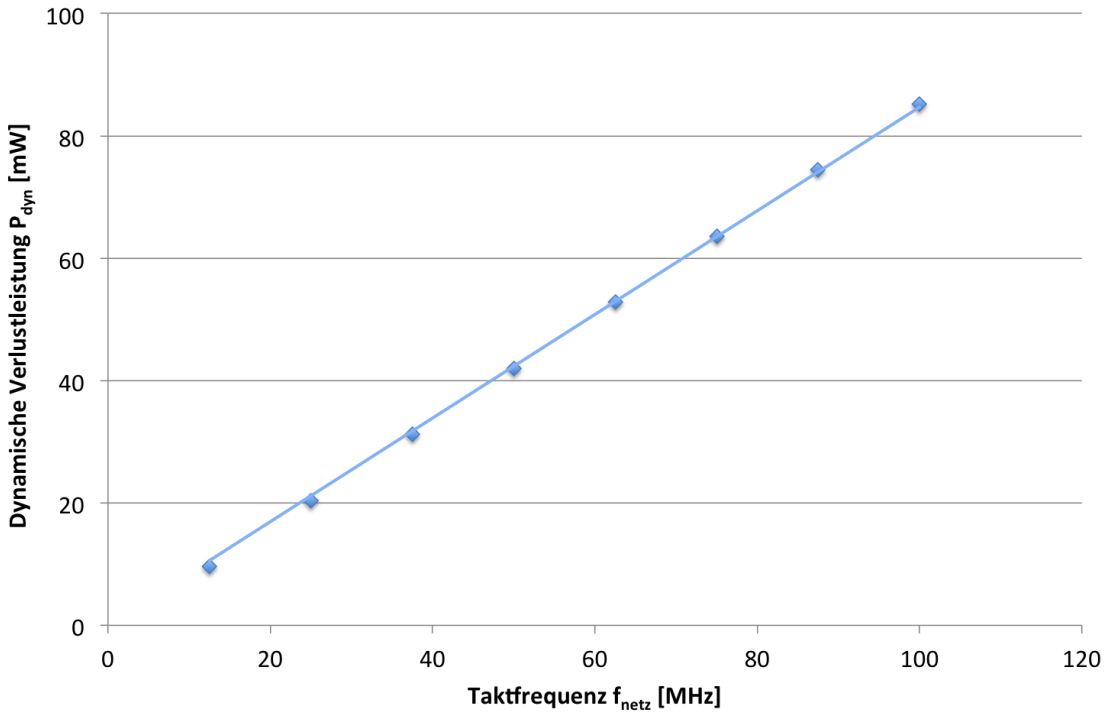


Abbildung 6.12: Abhängigkeit der dynamischen Verlustleistung des Verbindungsnetzwerkes der Größe 8x8 von dessen Taktfrequenz am Beispiel des Virtex-4 FPGA

Taktfrequenz f_{netz} des Verbindungsnetzwerks. Sehr gut zu erkennen ist der lineare Verlauf der Messpunkte entlang der eingezeichneten Regressionsgeraden. Aus der Steigung der Regressionsgeraden ergibt sich eine frequenzabhängige dynamische Verlustleistung $P_{dyn}(f_{netz}) = 0{,}85\ mW/MHz$. Neben der Taktfrequenz resultieren weitere Parameter aus der Größe und der Topologie des Verbindungsnetzwerks sowie dem verwendeten Scheduler. Auch die Größe der für die asynchrone Kommunikation benutzten FIFO-Puffer wird hier vermutlich eine Rolle spielen.

6.2.6 Einfluss des Prozessordesigns

Einfluss der Prozessorkomponenten. Softprozessoren besitzen kein festes Design, sondern können je nach Anwendung unterschiedlich gestaltet werden. So können z.B. zusätzliche Rechenwerke implementiert werden, um die Rechenleistung des Prozessors effektiv zu steigern. Je nach Ausbaustufe werden dadurch zusätzliche Ressourcen im FPGA belegt[17], was auch entsprechende Auswirkungen auf den Energiebedarf des Prozessors haben muss. Die **Abbildung 6.13** zeigt den Anteil verschiedener Komponenten eines Softprozessors als Ergebnis von Messungen, die an einem Softprozessor vom Typ Xilinx MicroBlaze in voller Ausbaustufe durch-

[17] vgl. Abbildung 4.13 : Anteiliger Flächenbedarf im Softprozessor, S. 67

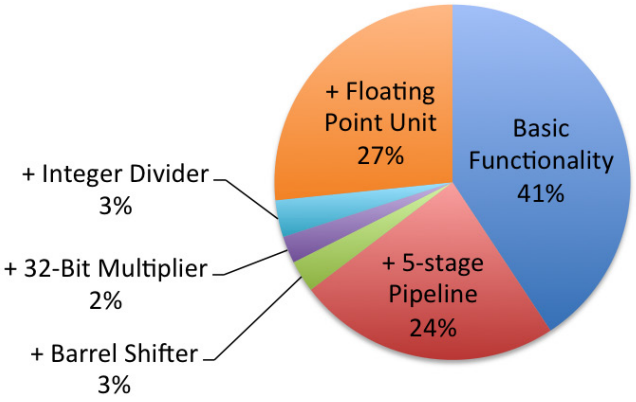


Abbildung 6.13: Anteiliger Energiebedarf verschiedener Komponenten im Softprozessor am Beispiel des Xilinx MicroBlaze

geführt wurden[18]. Dabei ist gut zu erkennen, dass fast 60 % des Energiebedarfs durch optionale leistungssteigernde Komponenten verbraucht wird. Eine Optimierung des Designs jedes einzelnen Prozessors ist deshalb ein wesentlicher Faktor bei der Optimierung der Verlustleistung im Rechensystem. Die größten Anteile entfallen auf die FPU (27 %) und die Befehlspipeline (24 %). Nicht so sehr ins Gewicht fallen dagegen Komponenten wie Barrel Shifter, Multiplizierer und Dividierer.

6.2.7 Zusammenfassung

Fazit. In diesem Abschnitt wurden eine Reihe von Messungen an verschiedenen FPGAs durchgeführt, um den Einfluss unterschiedlichster Parameter auf die dynamische Verlustleistung in einem FPGA-basierten Multiprozessorsystem zu untersuchen. Dabei konnten zunächst die theoretischen Grundlagen aus dem Abschnitt 6.1.2 Dynamische Verlustleistung, S. 174 ff. in Bezug auf den linearen Einfluss der Taktfrequenz durch Messergebnisse mit Zahlen belegt werden. Zusätzlich geben die Messungen aber auch Aufschluss über den Einfluss weiterer Parameter des Systems, wie z.B. die Anzahl der Softprozessoren, die Größe der Lokalspeicher oder das Prozessordesign. Insgesamt wurde in den Messungen eine ganze Reihe von Parametern nachgewiesen, welche die Verlustleistung des Rechensystems in irgendeiner Form beeinflussen. Darüber hinaus geben die Messdaten Auskunft über die Größenordnung des möglichen Einsparpotentials, welches nun im nachfolgenden Abschnitt 6.3 zu bewerten ist.

[18] vgl. B.6 Verlustleistung in Abhängigkeit vom Prozessordesign, S. 297 ff.

6.3 Optimierung der Verlustleistung im MPSoC

Inhalt. Space-Sharing bietet die Möglichkeit, FPGA-basierte Multiprozessorsysteme (MPSoC) gezielt auf die Rechenanforderungen der jeweiligen Softwareanwendung anzupassen. In diesem Abschnitt wird gezeigt, wie die Verlustleistung von Echtzeit-Rechensystemen optimiert und so der Energieverbrauch eines Steuergerätes im Auto unter Beibehaltung aller Echtzeitanforderungen minimiert werden kann. Die Grundlage hierfür bieten die Erkenntnisse aus den in Abschnitt 6.2 Verlustleistungen im FPGA-basierten Multiprozessorsystem, S. 177 ff. vorgestellten Ergebnisse der am FPGA durchgeführten Messungen.

Nutzen einer Leistungsoptimierung. Da ein Rechensystem die zugeführte elektrische Energie in keiner Weise speichern oder in mechanische Arbeit wandeln kann, geht praktisch die gesamte zugeführte Energie im System „verloren". Die Verluste werden direkt in Wärme umgesetzt, wodurch neben der verringerten Reichweite respektive Laufzeit mobiler Systeme noch weitere unerwünschte Effekte auftreten. Zum einen wird unter Umständen weitere Energie zur aktiven Kühlung der Bauteile benötigt, um eine Überhitzung zu vermeiden. Zum zweiten wird die Verfügbarkeit des Systems negativ beeinflusst, da nach dem Arrhenius-Modell im Bereich von 70° C bis 120 °C eine um 10 °C niedrigere Temperatur etwa eine Halbierung der Ausfallrate der Bauelemente zur Folge hat[19]. Zudem steigt die statische Verlustleistung mit der Temperatur der Halbleiter signifikant an. So erreicht die Leckage zwischen 25 °C und 85 °C das Zwei- bis Dreifache des Ausgangswertes[20]. Eine Optimierung der dynamischen Verlustleistung trägt somit indirekt auch zur Verringerung der statischen Verlustleistung bei.

6.3.1 Partitionierung der Software

Partitionierung der benötigten Rechenleistung. Der große Unterschied des FPGA-basierten Space-Sharing Modells gegenüber dem konventionellen Time Sharing liegt darin, dass im Space-Sharing für jede Task ein eigener Prozessor zugewiesen wird. Durch die softwarebezogene Partitionierung des Rechensystems entstehen wiederum neue Freiheitsgrade, welche neben dem Aspekt der Skalierbarkeit echtzeitfähiger Rechensysteme auch die Optimierung der Rechnerarchitektur im Hinblick auf die dynamische Verlustleistung des Rechensystems bzw. des Steuerge-

[19] vgl. [Cu07, S. 2]

[20] vgl. [Kl05, S. 5]

rätes erlauben. Da in einem Steuergerät die Laufzeiteigenschaften der Tasks bereits im Vorfeld bekannt sind, können verlustleistungsbeeinflussende Parameter des Multiprozessorsystems, wie z.B. Speichergröße oder Rechengeschwindigkeit der Prozessoren, für jede Task separat definiert werden. Die Gleichungen 6.8 bis 6.10 zeigen die Einflüsse verschiedener Parameter auf die Verlustleistung, wie sie am Virtex-4 FPGA gemessen wurden.

Prozessoren
$$P_{dyn} = 1 \frac{mW}{MHz} \tag{6.8}$$

Speicher
$$P_{dyn} = 0{,}01 \frac{mW}{KiB \cdot MHz} \tag{6.9}$$

Netzwerk
$$P_{dyn} = 0{,}85 \frac{mW}{MHz} \tag{6.10}$$

Beispiel. Auf Grundlage der am Virtex-4 FPGA ermittelten Werte zeigt die **Abbildung 6.14** die Verlustleistungen in einem beispielhaften MPSoC mit acht PMMs, bestehend aus identischen aber unterschiedlich getakteten Softprozessoren mit unterschiedlich großen Lokalspeichern, welche über ein asynchrones Verbindungsnetzwerk gekoppelt sind. Interessant ist an diesem Modell die Tatsache, wie verschieden die dynamischen Verlustleistungen der einzelnen Komponenten in Abhängigkeit von der Taktfrequenz oder der Speichergröße in einem MPSoC trotz der im Prinzip identischen Rechnerarchitektur ausfallen können.

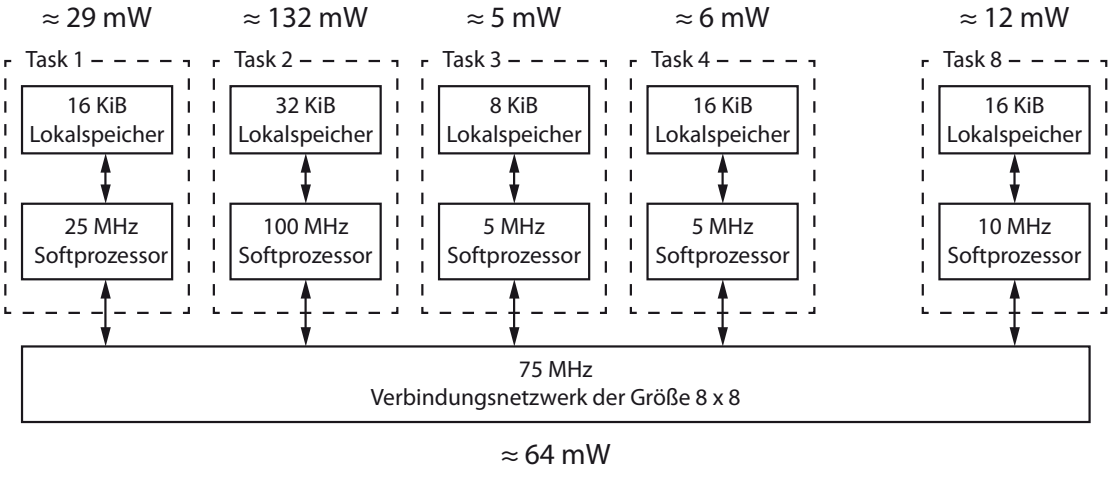


Abbildung 6.14: Verlustleistungen im FPGA-basierten Multiprozessorsystem

6.3.2 GALS-Design

Zeitliche Entkopplung. Auf der Anwenderseite kann die dynamische Verlustleistung des Multiprozessorsystems durch eine Optimierung der Taktfrequenz einzelner Komponenten im System minimiert werden. Da die einzelnen Tasks unterschiedliche Anforderungen bezüglich ihrer Bearbeitungsgeschwindigkeit besitzen, kann bei Space-Sharing aufgrund der statischen Zuordnung und der daraus resultierenden zeitlichen Entkopplung[21] jeder Prozessor mit einer individuellen Frequenz getaktet werden. Dazu müssen die Prozessoren innerhalb des MPSoCs zeitlich voneinander entkoppelt werden. Dies geschieht durch ein sogenanntes GALS-Design[22] des MPSoC.

Global asynchron - lokal synchron. GALS steht als Begriff für die Kombination aus getakteter (synchroner) und ungetakteter (asynchroner) Logik in einem Chip-Design. Der Hintergrund für ein solches Design bestand ursprünglich darin, dass einzelne ungünstige Verbindungen auf dem Chip die maximale Taktfrequenz des Systems beeinträchtigen können. Im GALS-Design wird das System in einzelne Module geteilt, welche intern synchron getaktet sind. Die externe Verbindung zwischen den Modulen erfolgt asynchron, so dass die Module zeitlich unabhängig voneinander getaktet werden können[23]. Das Prinzip des GALS-Design lässt sich auch auf Space-Sharing und die Architektur des FPGA-basierten MPSoCs übertragen. Jedes Modul besteht aus einem Prozessor inklusive Lokalspeicher plus asynchroner Netzwerkanbindung und kann unabhängig von den anderen Prozessoren getaktet werden. Die modulinterne Taktfrequenz wird dabei von der Ausführungszeit der jeweiligen Task bestimmt.

6.3.3 Dynamische Anpassung der Prozessorleistung

Inhalt. Nicht jede Task muss mit der gleichen Geschwindigkeit bearbeitet werden, und auch innerhalb einer Task finden sich Programmabschnitte, die unterschiedlich schnell verarbeitet werden müssen. So kann eine Warteschleife im Programm deutlich langsamer ablaufen als etwa ein Algorithmus, dessen Ergebnis schnellstmöglich zur Verfügung stehen soll. Um die dynamische Verlustleistung im Prozessor möglichst gering zu halten, soll die Prozessorleistung dynamisch an

[21] vgl. 4.3.2 Zeitliche Isolation durch Space-Sharing, S. 56 ff.

[22] engl. *globally asynchronous, locally synchronous*

[23] vgl. [Ro03]

den auszuführenden Programmabschnitt angepasst werden. Für diesen Zweck sollen die nachfolgenden Ausführungen entsprechende Maßnahmen aufzeigen.

6.3.3.1 Variable Steuerung des Prozessortaktes

Busgesteuerte Taktkontrolle. Um die dynamische Verlustleistung zur Laufzeit für jeden Programmabschnitt zu optimieren, muss der Prozessor selbst in der Lage sein, seinen eigenen Prozessortakt zu steuern. Da die Prozessoren im MPSoC unabhängig voneinander laufen, muss für jeden Prozessor ein eigener Prozessortakt generiert werden. Die **Abbildung 6.15** zeigt, wie der Prozessortakt für einen einzelnen Softprozessor im MPSoC erzeugt wird. Das Taktsignal wird zunächst von einem externen Oszillator generiert und in den FPGA gespeist. Aus dem externen Taktsignal erzeugt der FPGA-interne Taktgenerator ein global verfügbares Taktsignal fester Frequenz. Die Taktfrequenz des globalen Taktsignals stellt die maximale Frequenz dar, mit welcher der Softprozessor später arbeiten kann. Das globale Taktsignal wird nun an einen prozessoreigenen Taktteiler geleitet, der mithilfe eines Parameters unterschiedlich schnelle Taktraten an den Prozessor ausgeben kann. Der Teiler kann fest eingestellt oder, wie in der Abbildung 6.15 dargestellt, von einer externen Quelle vorgegeben werden. In dem dargestellten Beispiel kann der Prozessor über den Prozessorbus zur Laufzeit selbst festlegen, wie schnell er während der Programmausführung getaktet werden muss[24].

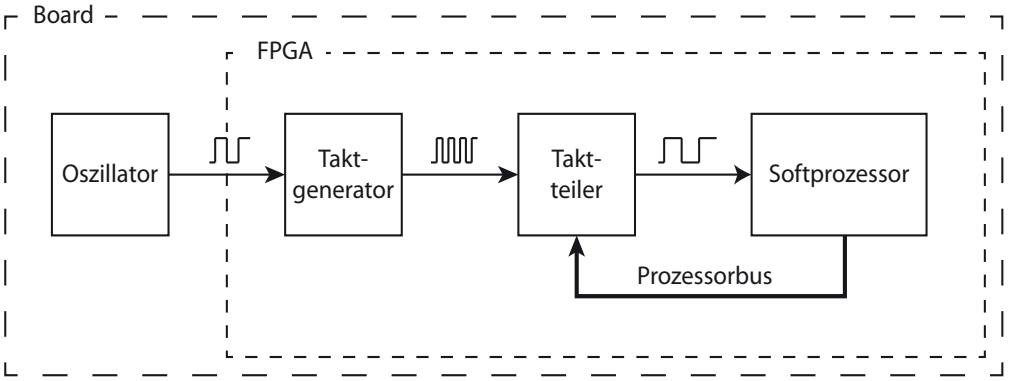


Abbildung 6.15: Busgesteuerte Taktkontrolle eines Softprozessors im FPGA

[24] weiterführende Informationen zur Implementierung der variablen Taktsteuerung im FPGA befinden sich im Abschnitt A.5.1 Variable Steuerung des Prozessortaktes, S. 274 ff. dieser Arbeit

6.3.3.2 Explizite Deklaration der Taktrate

Explizite Deklaration. Bei der expliziten Deklaration der Taktrate kann das Programm durch einen entsprechenden Befehl den Teiler neu setzen. Der Vorgang beschränkt sich auf eine kurze Zeile im Programm. Der neue Teiler wird dann über den Prozessorbus an den Taktteiler ausgegeben. Der Vorteil besteht darin, dass an der Entwicklungsumgebung keine Änderungen erforderlich sind. Der Nachteil besteht allerdings darin, dass der Programmierer abschätzen muss, an welcher Stelle im Programm wie viel Rechenleistung benötigt wird. Eine genaue Analyse der zeitlichen Programmausführung kann wieder erst zur Laufzeit erfolgen.

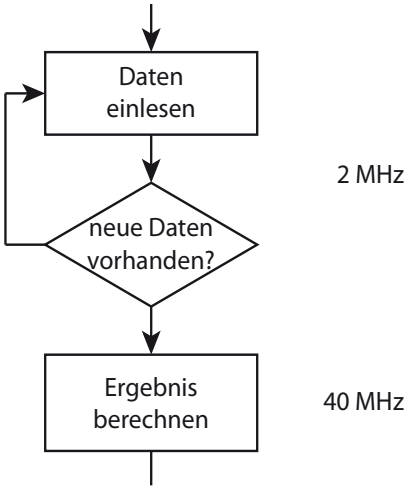


Abbildung 6.16: Programmbeispiel mit unterschiedlichen Taktraten

Beispiel. Die **Abbildung 6.16** zeigt an einem Programmbeispiel, wie die Zuweisung unterschiedlicher Taktraten innerhalb eines Programms erfolgen kann. In diesem Beispiel sollen Daten eingelesen und anschließend in einer Berechnungsroutine weiterverarbeitet werden. Für das Einlesen neuer Programmdaten werden deutlich weniger Befehle benötigt als für die nachfolgende Berechnung, weshalb hier mit einer niedrigeren Taktrate gearbeitet werden kann. Sobald neue Daten vorhanden sind, wird die Taktrate von 2 MHz auf 40 MHz erhöht und die Berechnung durchgeführt. Ohne eine dynamische Zuweisung der Taktrate müsste der Prozessor die ganze Zeit mit mindestens 40 MHz laufen. Die dynamische Absenkung der Taktrate auf 2 MHz zum Zeitpunkt des Wartens auf neue Daten reduziert die dynamische Verlustleistung in dieser Zeit auf 5 % von dem Wert, der während der nachfolgenden Berechnungsroutine aufgenommen wird.

Listing 6.1: Explizite Deklaration der Taktrate

```
clock_rate(2);    // Taktrate = 2 MHz
while (new_data = 0) new_data = check_new_data(&data);
clock_rate(40);    // Taktrate = 40 MHz
value = calc_value(&data);
clock_rate(2);    // Taktrate = 2 MHz
```

Zuweisung der Taktrate. Das **Listing 6.1** zeigt, wie die explizite Deklaration in dem Programmbeispiel aus Abbildung 6.16 aussieht. Die Zuweisung erfolgt durch einen einfachen Aufruf einer Routine, welche die neue Taktrate über den Prozessorbus an den Taktteiler übergibt.

6.3.3.3 Implizite Deklaration der Taktrate

Implizite Deklaration. Bei der impliziten Deklaration der Taktrate gibt der Programmierer Zeitmarken im Programm an. Diese Zeitmarken definieren, in welcher Zeit ein bestimmter Programmabschnitt abgearbeitet werden muss. Aus der Anzahl der zur Ausführung benötigten Prozessortakte und dem vorgebenen Zeitfenster kann dann die Taktrate für den jeweiligen Programmabschnitt automatisch bestimmt werden. Der Vorteil liegt darin, dass die Einhaltung der Zeitvorgaben im Programm bereits vor der Laufzeit gesichert werden kann. Der Nachteil besteht darin, dass Änderungen an der Entwicklungsumgebung vorgenommen werden müssen, da die Zeitangaben ausgewertet und später als Taktvorgaben im kompilierten Programm implementiert werden.

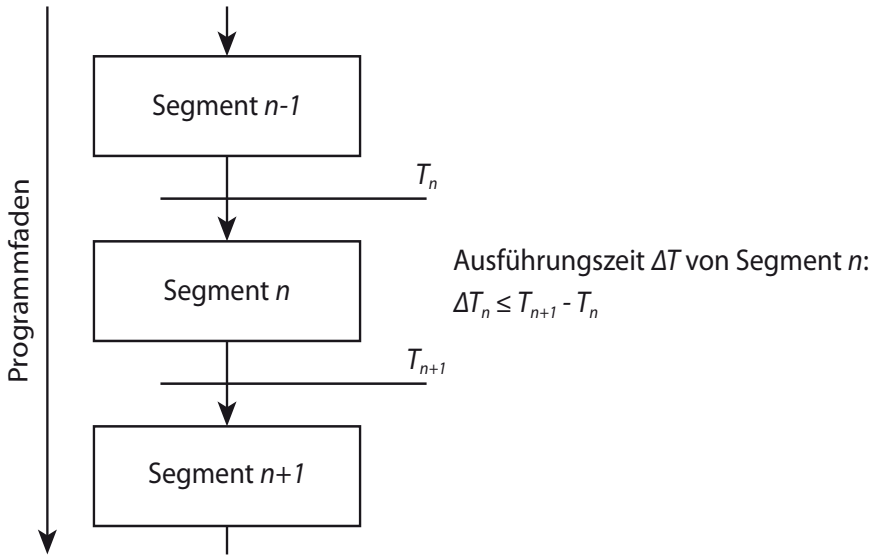


Abbildung 6.17: Segmentierung des Programmes durch Zeitmarken

Segmentierung durch Zeitmarken. Die **Abbildung 6.17** zeigt die zeitliche Segmentierung eines Programmfadens. Die Kennzeichnung der Segmente erfolgt durch Zeitmarken. Die Zeit, die zur Ausführung eines Segmentes n zur Verfügung steht, ergibt sich aus dem zeitlichen Abstand ΔT_n der Zeitmarken T_n am Beginn und T_{n+1} am Ende jeden Segmentes.

Ermittlung der erforderlichen Taktrate. Die Zeit ΔT_n für ein Segment n im Programm wird durch die Angabe von Zeitmarken am Beginn und am Ende des Segmentes bestimmt. Jedes Segment beinhaltet eine definierte Anzahl *i* an Befehlen zur Ausführung, wobei jeder Befehl eine feste Anzahl an Taktzyklen benötigt. Die Anzahl der Taktzyklen kann je nach Ausbaustufe des Prozessors (z.B. 3- oder 5-Stufen Pipeline) und Art des Befehls zwischen 1 (bei Integeraddition) und 30 (bei Fließkommadivision) betragen[25]. Eine Ausnahme stellen Sprungbefehle dar. Je nachdem, ob der Sprung genommen wird, dauert die Ausführung eines Sprungbefehls zwischen 1 und 3 Taktzyklen aufgrund der anschließenden Invalidierung der ersten Pipelinestufen[26]. Da Sprunganweisungen nicht zu 100 Prozent vorhersagbar sind, muss bei der Ermittlung der erforderlichen Taktrate immer von der maximalen Anzahl der Taktzyklen ausgegangen werden.

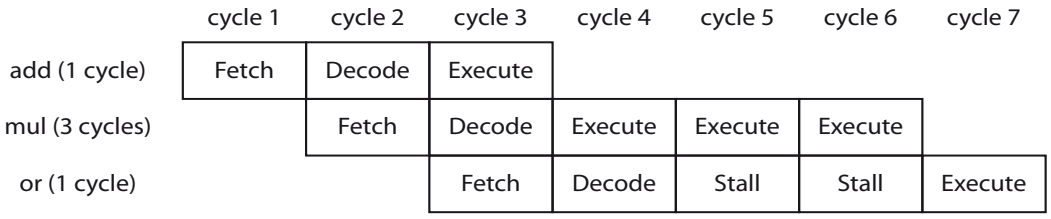


Abbildung 6.18: Befehlsausführung in einer 3 Stufen Pipeline

Die **Abbildung 6.18** zeigt die Ausführung verschiedener Befehle in einer kostenoptimierten 3-Stufen-Pipeline. Benötigt ein Befehl mehr als einen Taktzyklus zur Ausführung, so werden die nachfolgenden Befehle in der Pipeline zurückgehalten. Eine dynamische Befehlsausführung (*out-of-order execution*) besitzt der MicroBlaze nicht. Dadurch kann die Anzahl der erforderlichen Taktzyklen für ein Programmsegment einfach aufsummiert werden. Die erforderliche Taktrate des Prozessors f_{takt} für die Ausführung des Programms ergibt sich aus der Summe der Taktzyklen c pro Befehl und der Zeitschranke ΔT_n:

[25] vgl. [Xi08b, S. 103 ff.]

[26] vgl. [Xi08b, S. 44]

$$f_{takt} = \frac{\sum_{k=1}^{i} c_k}{\Delta T_n} \tag{6.11}$$

Das übersetzte Segment wird durch einen Postprozessor analysiert und die Taktrate für das Segment entsprechend der Gleichung 6.11 bestimmt.

Listing 6.2: Implizite Deklaration der Taktrate durch Zeitmarken

```
/*! initiate */
while (new_data = 0) new_data = check_new_data(&data);
/*! event ≤ initiate + 20 */   // 40 Taktzyklen/20µs = 2 MHz
value = calc_value(&data);
/*! terminate ≤ event + 30 */   // 1200 Taktzyklen/30µs = 40 MHz
```

Setzen von Zeitmarken. Das **Listing 6.2** zeigt in einem Beispiel, wie das Setzen von Zeitmarken im Programmcode aussieht. Die Anweisungen für den Postprozessor werden im Programm als Kommentare angegeben. In dem dargestellten Beispiel soll das Segment zwischen den Zeitmarken „initiate“ und „event“ in 20 µs abgearbeitet werden. Da die Befehlsausführung 40 Taktzyklen dauert, muss der Prozessor in diesem Segment mit 2 MHz arbeiten. In dem darauffolgenden Segment mit den Zeitmarken „event“ und „terminate“ muss der Prozessor mit 40 MHz laufen, da an dieser Stelle 1200 Takte innerhab eines Zeitfensters von 30 µs erfolgen müssen.

6.3.4 Weiterführende Maßnahmen

Prozessorarchitektur. Unterschiedliche Anwendungen besitzen auch unterschiedliche Anforderungen an den Befehlssatz bzw. die Architektur des Prozessors. Durch den Einsatz von Softprozessoren ist es möglich, die Prozessorarchitektur bzw. die Rechenhardware im Rahmen eines Software-First-Designs[27] an die Anforderungen der jeweiligen Task anzupassen. Ein Beispiel hierfür wäre eine zusätzliche Gleitkommaeinheit (FPU) für mathematische Berechnungen. Spezielle Berechnungen wie FFT-Funktionen können durch anwendungsspezifische Koprozessoren parallel in Hardware ausgeführt werden. Mit den genannten Maßnahmen kann die Anzahl der Befehle bzw. die Anzahl der Prozessortakte für eine gegebene Aufgabe reduziert werden. Dadurch erfolgt effektiv eine Steigerung der Rechenleistung, was wiederum geringere Schaltfrequenzen und Schaltspannungen erlaubt.

[27] vgl. 4.5.1 System-Design, S. 60

Dem gegenüber stehen allerdings der zusätzliche Ressourcenverbrauch im FPGA[28] und der bis zu 2,5-fach erhöhte Energiebedarf durch zusätzliche Komponenten im Prozessordesign[29]. Eine detaillierte Beschreibung verschiedener Möglichkeiten zur Senkung der Verlustleistung durch eine geeignete Prozessorarchitektur befindet sich in [Pi96].

Spannungs- und Frequenzinseln. In einem GALS-Design[30] gibt es die Möglichkeit[31], gleichzeitig mit der Taktfrequenz auch die Taktspannung zu reduzieren oder nicht benötigte Teile des Chips (z.B. einzelne Prozessoren) spannungslos zu schalten. Der Vorteil der Taktspannung besteht gegenüber der Taktfrequenz darin, dass die Taktspannung quadratisch in die dynamische Verlustleistung eingeht. Darüber hinaus wird auch die statische Verlustleistung positiv beeinflusst. Der Effekt wäre hier also weitaus größer als bei der alleinigen Absenkung der Taktfrequenz. Eine Absenkung der Taktspannung in einem GALS ist allerdings nur bei entsprechender Hardware möglich. Eine ausführlichere Beschreibung, wie ein solches SoC-Design aussehen müsste, befindet sich in [La02]. Eine Realisierung der Absenkung von Taktspannung und Taktfrequenz in einem GALS-Design mit konkreten Messergebnissen befindet sich in [Ni05].

6.4 Zusammenfassung

Energieverbrauch. Der Energieverbrauch eines Multiprozessorsystems (MPSoC) wird primär durch dessen Verlustleistung bestimmt. Die Verlustleistung gliedert sich je nach Ursache in einen statischen und einen dynamischen Teil (Gleichung 6.12).

$$P_{verlust} = \sum_{i=1}^{m} \left(C \cdot U^2 \cdot f \right)_i + \sum_{j=1}^{n} \left(U \cdot I \right)_j \qquad (6.12)$$

Statische Verluste. Der statische Teil der Verlustleistung ist taktunabhängig. Er wird durch Leckströme und Unterschwellspannungsströme im Halbleitermaterial verursacht. Die fortschreitende Miniaturisierung der Halbleiterstrukturen bedingt geringere Schichtdicken des dotierten Materials, wodurch der Anteil der statischen

[28] vgl. Abbildung 4.13 : Anteiliger Flächenbedarf im Softprozessor, S. 67

[29] vgl. Abbildung 6.13 : Anteiliger Energiebedarf verschiedener Komponenten im Softprozessor am Beispiel des Xilinx MicroBlaze, S. 187

[30] vgl. 6.3.2 GALS-Design, S. 190

[31] entsprechende Hardware vorausgesetzt

Verlustleistung an der Gesamtverlustleistung in der Zukunft sehr wahrscheinlich weiter wachsen wird. Für den Anwender bietet sich jedoch keine Möglichkeit, die statische Verlustleistung zu beeinflussen.

Dynamische Verluste. Der dynamische Teil der Verlustleistung ist abhängig von mehreren Faktoren: der Schaltkapazität, der Schaltspannung sowie der Schaltfrequenz. Schaltkapazitäten werden durch den Herstellungsprozess, aber auch durch Place&Route-Werkzeuge bei der Synthese der Schaltungen in Hardware beeinflusst. Die Schaltspannung bietet zwar das größte Einsparpotential, sie kann aber nur bei einem entsprechenden SoC-Design für Teile des Chips geändert werden. Bei handelsüblichen FPGAs bietet lediglich die Schaltfrequenz effektive Möglichkeiten zur Optimierung des Energieverbrauchs im MPSoC.

Optimierung des Energieverbrauchs durch Space-Sharing. Durch die softwareorientierte Partitionierung der erforderlichen Rechenleistung auf mehrere Prozessoren kann bei Space-Sharing jeder Prozessor so schnell getaktet werden, wie es die jeweilige Anwendung erfordert. Zusätzlich können im FPGA-basierten Software-First-Design weitere Parameter, wie z.B. die Größe der Lokalspeicher und die Prozessorarchitektur, an die Erfordernisse der Software angepasst werden. Darüberhinaus kann die Taktfrequenz von einzelnen Prozessoren durch geeignete Taktteiler-Bausteine (*gated clocks*) dynamisch zur Laufzeit angepasst werden. Dadurch wird es möglich, die dynamische Verlustleistung des Rechensystems auf ein Minimum zu senken.

Kapitel 7 - Integration in AUTOSAR

Inhalt. In diesem Kapitel geht es um die Integration von *Space-Sharing* in das Konzept von AUTOSAR, welches derzeit als zukünftiger de-facto Standard bei der Entwicklung automobiler Steuergeräte gehandelt wird. Dazu wird auf die Bedeutung des Schichtenmodells von AUTOSAR bei der Rezentralisierung der Steuergerätesoftware eingegangen. Darüber hinaus wird gezeigt, wie das *Software-First-Design*[1] die bestehende AUTOSAR-Methodik beeinflusst, welche auf dem Ansatz des Hardware-First-Designs basiert.

7.1 Grundlagen

Inhalt. In diesem Abschnitt wird kurz auf die wichtigsten Grundlagen von AUTOSAR eingegangen, bevor in den nachfolgenden Abschnitten die Details zur Integration von Space-Sharing in das Konzept von AUTOSAR betrachtet werden.

7.1.1 Bedeutung

Entstehung und Ziele. Die herstellerübergreifende Initiative AUTOSAR[2] wurde im Jahre 2003 von damals 10 Core Partnern ins Leben gerufen. Inzwischen vereint AUTOSAR viele namhafte Unternehmen aus der Hersteller- und Zulieferindustrie als Core Partner (9), Premium Members (55), Associate Members (83) und Development Members (6)[3]. Das erklärte Ziel dieser Initiative ist es, eine herstellerübergeifende, offene und standardisierte Softwarearchitektur für automobile Steuergeräte zu schaffen. Die Spezifikation von AUTOSAR soll helfen, den Entwicklungsprozess von automobilen Steuergeräten bei Herstellern und Zulieferern an die zukünftigen Herausforderungen der Automobilindustrie anzupassen.

Bedeutung. Es gibt zwei wesentliche Gründe um die Spezifikation von AUTOSAR im Rahmen dieser Arbeit näher zu betrachten: Der erste Grund besteht darin, dass die Einführung von AUTOSAR bei der Entwicklung automobiler Steuergeräte derzeit eines der wichtigsten Themen in den Diskussionen der Automobilindustrie darstellt. Dabei gilt es zum jetzigen Zeitpunkt als sicher, dass sich AUTOSAR

[1] vgl. 4.5.1 System-Design, S. 60 ff.

[2] Abkürzung für AUTomotive Open System ARchitecture

[3] Stand vom Mai 2009

in der nahen Zukunft als Standard für die Entwicklung automobiler Steuergeräte etablieren wird. Der zweite Grund besteht in der geschichteten Architektur der AUTOSAR-Software. Die Trennung der Funktionssoftware von der Hardware der Steuergeräte bildet einen grundlegenden Vorteil bei deren Rezentralisierung.

Warum AUTOSAR? Wie bereits in der Einleitung zu dieser Arbeit beschrieben, erreichen Elektronik und Software im Automobil nach dem heutigen Stand eine dramatisch wachsende Komplexität. Die Schwierigkeiten, welche dieses komplexe System mit sich bringt, werden zusätzlich verstärkt durch eine verteilte Topologie von Komponenten unterschiedlicher Hersteller im Fahrzeug, welche über die vergangenen Jahrzehnte historisch gewachsen ist[4]. Die vorherrschende konstruktive Sicht eines verteilten Systems, bei dem Teilsysteme oder sogar einzelne Funktionen im Fahrzeug ein eigenes Steuergerät benötigen, führt zwangsläufig zu einem steuergerätezentrierten Entwurf des Fahrzeugs, dessen Umsetzung am hohen Vernetzungsgrad scheitert[5]. Für die Rezentralisierung der Steuergeräte soll nun die zu realisierende Funktionalität im Fahrzeug in den Mittelpunkt gerückt werden. Dies bedingt jedoch einen architekturbasierten Ansatz, welcher ohne die feste Zuordnung von Funktionssoftware und Steuergeräte-Hardware auskommt. Mit der Einführung von AUTOSAR scheint dieses Ziel erreichbarer denn je. Mit AUTOSAR bietet sich eine geeignete Basis, um die bestehende Steuergerätesoftware in *ConPar* zu rezentralisieren. Allerdings muss dafür zum Teil von der derzeitigen Form der AUTOSAR-Spezifikation, z.B. in der AUTOSAR-Methodik, abgewichen werden. Unabhängig davon bleibt jedoch das primäre Ziel von AUTOSAR, die dramatisch gestiegene Komplexität von Elektronik und Software im Automobil beherrschbar zu machen, ohne Einschränkungen bestehen.

7.1.2 Dokumentation

Spezifikation von AUTOSAR. Dieses Kapitel bezieht sich auf die Spezifikation von AUTOSAR in den Versionen 3.1 und 4.0, deren vollständige Dokumentation auf der offiziellen Webseite[6] verfügbar ist. Die Spezifikation von AUTOSAR enthält weitere umfangreiche Informationen zu AUTOSAR, die weit über die vorliegende Arbeit hinausgehen, sowie ein Glossar zu den verwendeten Begriffen [AS09c].

[4] vgl. 2.1.1 Dezentrale Topologie, S. 9 f.

[5] vgl. [Ki09, S. 41 f.]

[6] offizielle Webseite von AUTOSAR unter www.autosar.org

7.2 Softwarearchitektur

Schichtenmodell. Die Architektur von AUTOSAR beruht auf einem geschichteten Softwaremodell, wie es in der **Abbildung 7.1** dargestellt ist. Durch die Einführung der verschiedenen Schichten und der Spezifikation der Schnittstellen zwischen den einzelnen Schichten kann die hardwareunabhängige Funktionssoftware auf der Anwendungsebene von der darunterliegenden Hardware bzw. von der hardwarespezifischen Software getrennt werden.

Wiederverwendbarkeit. Die Wiederverwendbarkeit von Software spielt in der modernen Steuergeräteentwicklung eine ganz wesentliche Rolle und gehört damit zu den primären Zielen von AUTOSAR. Vor dem Hintergrund des rapide wachsenden Anteils der Automobilelektronik[7] am Fahrzeug bei gleichzeitig kürzer werdenden Modelllaufzeiten[8] ist die Wiederverwendung von bereits entwickeltem und erfolgreich eingesetztem Programmcode eine sehr effiziente Lösung, um Entwicklungszeit und -kosten zu sparen. Zudem ist wiederverwendeter Code i.d.R. bereits ausreichend getestet, so dass aufwändige Testverfahren, wie sie bei völlig neu entwickelter Software unumgänglich sind, dezimiert werden können.

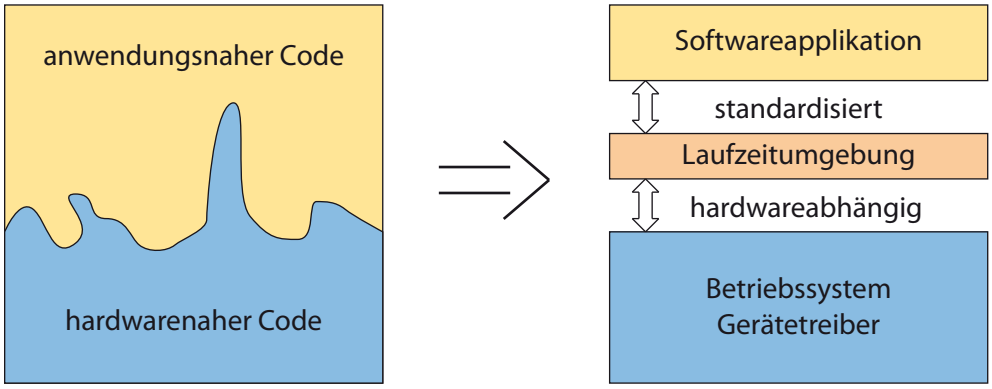


Abbildung 7.1: Abstraktion der Hardware in AUTOSAR[9]

Portabilität. Die Wiederverwendbarkeit von Funktionssoftware ist nur möglich, wenn diese auf verschiedene Hardwareplattformen portiert werden kann. Zu diesem Zweck stellt AUTOSAR eine Laufzeitumgebung[10] zur Verfügung, deren

[7] vgl. Abbildung 1.2: Steigerung des Anteils der Elektronik am Wert des Fahrzeug, S. 5

[8] vgl. Abbildung 2.3: Verringerung der Laufzeiten von Fahrzeugmodellen, S. 17

[9] grafische Darstellung nach [Ki09, S. 41]

[10] engl.: Run-Time Environment (RTE)

Schnittstellen zur darüberliegenden Funktionssoftware standardisiert sind (vgl. Abbildung 7.1b). Unterhalb der Laufzeitumgebung RTE befindet sich die Basissoftware, welche der Laufzeitumgebung die Dienste der zugrundeliegenden Hardware, z.B. Kommunikations- oder Speicherdienste, zur Verfügung stellt. Die Abstraktion der hardwarenahen Schichten durch standardisierte Schnittstellen zur Anwendungsebene erlaubt die Portierung bereits bestehender und getesteter Funktionssoftware auf nachfolgende Generationen von Steuergeräten. Für *ConPar* bedeutet dies, dass die Funktionalität der Steuergeräte auf den Echtzeitparallelrechner portiert werden kann, sofern der Rechner auf das Schichtenmodell von AUTOSAR abstrahiert. Eine Emulation ganzer Steuergeräte inklusive deren Hardware, wie es für die Portierung klassischer Steuergerätesoftware notwendig wäre, wird durch die Einführung des Schichtenmodells von AUTOSAR obsolet.

Methodik. In der AUTOSAR-Methodik sind standardisierte Formate für den Austausch von Informationen innerhalb der Werkzeugkette (*work-flow*) spezifiziert. Dies erlaubt den automatisierten Einsatz verschiedener Werkzeuge, aber auch die Portierung von Arbeitsprodukten innerhalb der Werkzeugkette. Darüber hinaus ist der Produktfluss nebst den darin enthaltenen Arbeitsprodukten spezifiziert. Aufgrund des Einsatzes von Space-Sharing und dem daraus resultierenden Software-First-Design[11] muss die Methodik von AUTOSAR für ConPar entsprechend modifziert werden[12].

Spezifikation der Schichten. Die **Abbildung 7.2** zeigt den Aufbau des Softwareschichtenmodells in stark vereinfachter Form, wie er in der Spezifikation von AUTOSAR enthalten ist[13]. Nachfolgend soll die Bedeutung des Schichtenmodells

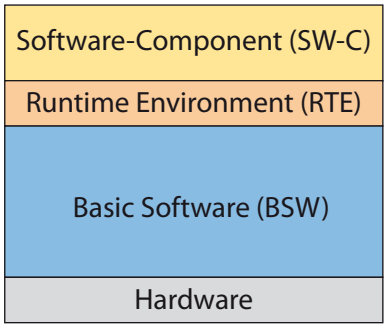


Abbildung 7.2: AUTOSAR Schichtenmodell

[11] vgl. 4.5.1 System-Design, S. 60 ff.

[12] vgl. 7.5 AUTOSAR-Methodik, S. 217 ff.

[13] vgl. [AS09a, S. 68]

von AUTOSAR für die Softwareentwicklung nach dem AUTOSAR-Standard und darüber hinaus für die Rezentralisierung von Steuergeräten im Echtzeitparallelrechner *ConPar* erläutert werden.

7.2.1 Basic Software (BSW)

Aufgaben. Die *Basic Software* (BSW) befindet sich im Schichtenmodell zwischen der Laufzeitumgebung RTE und der Hardware des Steuergerätes, dem Mikrocontroller und dessen Peripherie. Die Aufgabe der Basic Software besteht im Wesentlichen darin, die Laufzeitumgebung von der Hardware zu abstrahieren und verschiedene Dienste der Hardware und des Betriebssystems, z.B. Task-Scheduling, Speichermanagement oder Kommunikationsdienste, bereitzustellen.

Aufbau. In der **Abbildung 7.3** ist der Aufbau der Basic Software grob dargestellt. Diese Schicht ist noch einmal unterteilt in die Diensteschicht (*Service Layer*), die ECU-Abstraktionsschicht (*ECU Abstraction Layer*) und die µC-Abstraktionsschicht (*Microcontroller Abstraction Layer*).

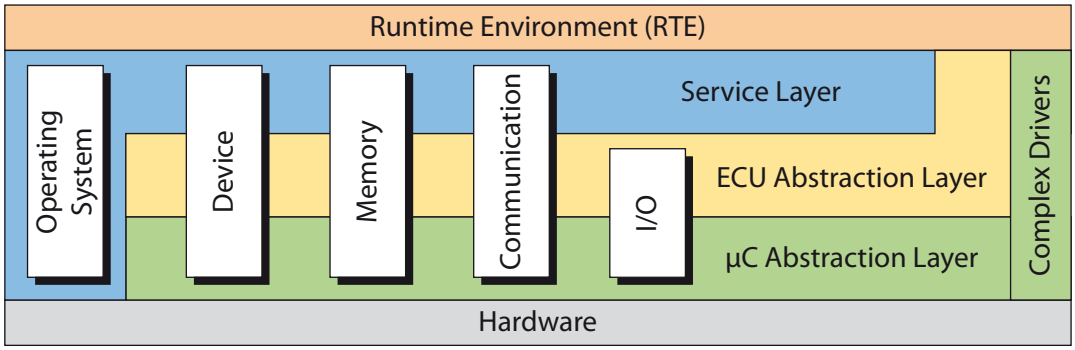


Abbildung 7.3: AUTOSAR Schichtenmodell der Basic Software

Microcontroller Abstraction Layer. Die µC-Abstraktionsschicht stellt die Unabhängigkeit der höheren Schichten von der Hardware sicher. Sie beinhaltet Low-Level Software, welche direkt auf die Hardware aufsetzt. Dazu zählen z.B. Gerätetreiber für Speicher- oder Kommunikationsbausteine.

ECU Abstraction Layer. Oberhalb der µC-Abstraktionsschicht befindet sich die ECU-Abstraktionsschicht. Diese Schicht stellt den darüberliegenden Diensten ein API für den Zugriff auf die benötigte Hardware zur Verfügung. Für die angebotenen Dienste bleibt die Hardware des Steuergerätes transparent. Eine Ausnahme bilden die komplexen Treiber (*Complex Drivers*). Vor dem Hintergrund nicht spezi-

fizierter oder zeitkritischer Schnittstellen kann eine Anwendung in AUTOSAR bei Bedarf auch direkt auf die Hardware zugreifen.

Service Layer. Die Diensteschicht ist die höchstgelegende Schicht der Basic Software. Sie stellt die Schnittstellen zur Laufzeitumgebung RTE bereit, über die Anwendungen (Funktionssoftware) auf das Betriebssystem, den Speicher und die Peripherie zugreifen oder mit den Anwendungen anderer Steuergeräte kommunizieren können.

Abhängigkeit von der Hardware. Da die Basic Software direkt auf der Hardware des Steuergerätes aufsetzt, wird diese Schicht dynamisch aus der ECU-Konfiguration generiert. Dies bedeutet, dass bei einer Änderung der Hardware des Steuergerätes, z.B. durch den Einsatz eines anderen Prozessors oder Bussystems, auch eine neue Version der Basic Software generiert werden muss. In der Methodik von AUTOSAR werden für diesen Zweck die spezifischen Informationen aus der Beschreibung der Hardware extrahiert und ausgewertet[14].

7.2.2 Software Component (SW-C)

Funktionssoftware. Auf der obersten Ebene des AUTOSAR Schichtenmodells, der Anwendungsebene, befinden sich die Softwarekomponenten (SW-Cs) des Steuergerätes (vgl. Abbildung 7.2). Die Softwarekomponenten (*Software Components*) bilden die vom Nutzer erlebbare Funktionalität im Fahrzeug ab und werden dementsprechend auch als Funktionssoftware bezeichnet.

Granularität der Funktionssoftware. Bei der Rezentralisierung von Steuergeräten im Echtzeitparallelrechner *ConPar* gilt es, das Prinzip des FPGA-basierten Space-Sharings[15] auf die zu rezentralisierende Funktionssoftware der im System verteilten Steuergeräte, den AUTOSAR-Softwarekomponenten, anzuwenden. Das heißt, dass die Funktionssoftware der Steuergeräte auf eine bestimmte Anzahl von Prozessoren im Echtzeitparallelrechner gemappt werden muss. Hierfür muss eine Design-Entscheidung über die Rechnerarchitektur getroffen werden, welche auf der Basis verschiedener Kriterien wie der Granularität der Funktionssoftware und dem Kommunikationsverhalten der Softwarekomponenten innerhalb und außerhalb des Steuergerätes beruht.

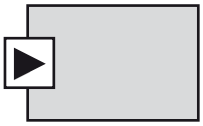
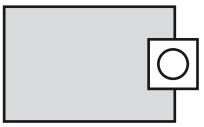
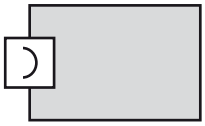
[14] vgl. 7.5 AUTOSAR-Methodik, S. 217 ff.

[15] vgl. Kapitel 4 - Space-Sharing, S. 46 ff.

7.2.2.1 Schnittstellen

Ports. Unter einem *Port* versteht AUTOSAR die Schnittstelle einer Softwarekomponente (SW-C) zur Interaktion mit anderen SW-Cs [AS09c, S. 61]. Ein Port kann Daten oder Operationen anbieten (*Provide Port*) oder die angebotenen Daten oder Operationen nutzen (*Require Port*). Darüber hinaus existieren auch Service Ports und Calibration Ports, um Einstellungen und Fehlerspeicher abzufragen oder Kalibrierungen vorzunehmen. Die **Tabelle 7.1** zeigt die grafische Notation der beiden wichtigsten Ports für die Sender/Receiver- und die Client/Server-Kommunikation zwischen Softwarekomponenten [AS09d, S. 16].

Tabelle 7.1: AUTOSAR Interface Ports[16]

Interface	Provide Port	Receive Port
Sender/Receiver		
Client/Server		

AUTOSAR Interfaces. Die Schnittstellen der Softwarekomponenten (SW-Cs) zur darunterliegenden Laufzeitumgebung (RTE) werden als AUTOSAR *Interfaces* bezeichnet (**Abbildung 7.4**). Ein Interface kann mehrere verschiedene Ports enthalten [AS09c, S. 71]. Die Verwendung standardisierter Ports als Interface erlaubt die Portierung von Softwarekomponenten auf AUTOSAR-konforme Steuergeräte verschiedener Steuergerätehersteller bzw. -generationen mit unterschiedlicher Hardware.

Rezentralisierung der SW-Cs. Durch die Einführung von standardisierten Schnittstellen zur Kommunikation auf der Anwendungsebene können AUTOSAR-konforme Softwarekomponenten (und damit die Funktionalität des Fahrzeugs) von den einzelnen Steuergeräten im verteilten System auf den Echtzeitparallelrechner *ConPar* portiert und somit rezentralisiert werden. Die grundlegende Bedingung für die Rezentralisierung der Steuergerätesoftware besteht darin, dass die darunterliegenden Schichten, die Laufzeitumgebung (RTE) und die Basissoftware (BSW), auf *ConPar* abgebildet werden können.

[16] eine ganze Reihe weiterer Ports sind in [AS09d, S. 16-19] aufgeführt

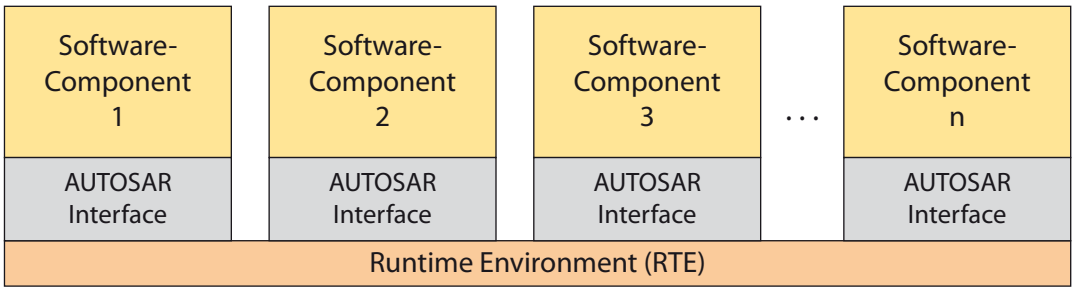


Abbildung 7.4: AUTOSAR Schichtenmodell der SW-Cs

7.2.2.2 Softwarestruktur

Mapping von SW-Cs auf Softprozessoren. Für die Rezentralisierung von Steuergeräten im Echtzeitparallelrechner ist die Granularität der Funktionssoftware ein ganz wesentlicher Parameter für die erforderliche Anzahl an Prozessoren und damit für die Größe und die Leistungsfähigkeit des Rechensystems[17]. Die Granularität wird in diesem Fall bestimmt durch die nebenläufige Struktur der Funktionssoftware. Es muss also geklärt werden, inwieweit die Funktionssoftware in nebenläufige Teile zerlegt und auf Softprozessoren gemappt werden kann.

Komposition. Softwarekomponenten können in AUTOSAR zu einer *Komposition* zusammengefasst werden. Kompositionen dienen zur hierarchischen Strukturierung von Programmen und zur Abstraktion von Funktionalitäten und werden daher mit einem Struktogramm in der rechten oberen Ecke markiert (**Abbildung 7.5**). Kompositionen können selbst Bestandteil weiterer Kompositionen und auch auf mehrere Steuergeräte verteilt sein [AS09d, S. 22 f.]. Kompositionen bilden demnach keine granulare Größe, sondern können in jedem Fall in ihre beinhalteten Softwarekomponenten zerlegt werden.

Abbildung 7.5: SW-C Komposition

Atomare Softwarekomponente. Softwarekomponenten, die selbst keine Komposition von Softwarekomponenten beinhalten, werden als *atomare Softwarekomponenten* bezeichnet und mit einem kleinen Rechteck in der rechten oberen Ecke

[17] vgl. 3.2.2 Amdahl'sches Gesetz, S. 39

markiert (**Abbildung 7.6**). Atomare Softwarekomponenten sind strukturell nicht teilbar und müssen in AUTOSAR auf ein einziges Steuergerät gemappt werden [AS09d, S. 30]. Insofern stellt die atomare SW-C zunächst eine feste granulare Größe bei der Rezentralisierung dar, da jede atomare Softwarekomponente auch auf einen eigenen Prozessor gemappt werden kann.

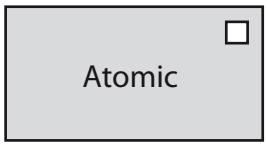


Abbildung 7.6: Atomare SW-C

Runnable Entity. Jede AUTOSAR Software-Komponente (SW-C) enthält mindestens einen Programmfaden in Form einer *Runnable Entity* (oder kurz: Runnable). Runnables können unabhängig von anderen Runnables ausgeführt werden und werden mit einem Pfeil in der rechten oberen Ecke markiert (**Abbildung 7.7**). Runnables enthalten eine Sequenz von Instruktionen und werden aus der Laufzeitumgebung RTE heraus gestartet, wobei jede Runnable ihren eigenen Startpunkt (*Entry Point*) bzw. Endpunkt (*Exit Point*) zur Programm bearbeitung besitzt [AS09c, S. 66]. Für Space-Sharing bedeutet dies, dass ein Runnable die kleinste atomare Softwarestruktur darstellt, um Nebenläufigkeiten im Programm zu erzeugen. Insofern sind die Voraussetzungen für eine nebenläufige Ausführung von Runnables einer SW-C auf verschiedenen Prozessoren als granulare Größe bei der Rezentralisiserung im Echtzeitparallelrechner gegeben. Die entscheidende Punkt besteht hier in der Kommunikation zwischen den Runnables innerhalb einer Softwarekomponente, welche im Unterschied zur SW-C-externen Kommunikation i.d.R. durch Interrunnable Variablen, d.h. durch einen gemeinsamen Speicher, realisiert wird [AS09a, S. 230].

Abbildung 7.7: SW-C Runnable

7.2.2.3 Prozessor-Mapping

Nebenläufige Einheiten. Das Konzept von Space-Sharing beruht darauf, Software in Nebenläufigkeit auf mehreren Prozessoren auszuführen. Das bedeutet für die

Rezentralisierung der verteilten Steuergeräte in einem Echtzeitparallelrechner, dass die Funktionssoftware der Steuergeräte in eine bestimmte Anzahl nebenläufiger Einheiten aufgeteilt und auf ebenso viele Prozessoren des Echtzeitparallelrechners gemappt werden muss. Nach den SW-Cs stellen Runnables die kleinste unteilbare Größe für das Mapping der Funktionssoftware auf die Prozessoren des Echtzeitparallelrechners dar. Die Größe nebenläufiger Einheiten[18] reicht von der Runnable Entity bis zur kompletten Funktionssoftware eines Steuergerätes. Die **Abbildung 7.8** zeigt das Mapping nebenläufiger Einheiten auf Prozessoren im Echtzeitparallelrechner am Beispiel einer stark vereinfacht dargestellten Spiegelverstellung.

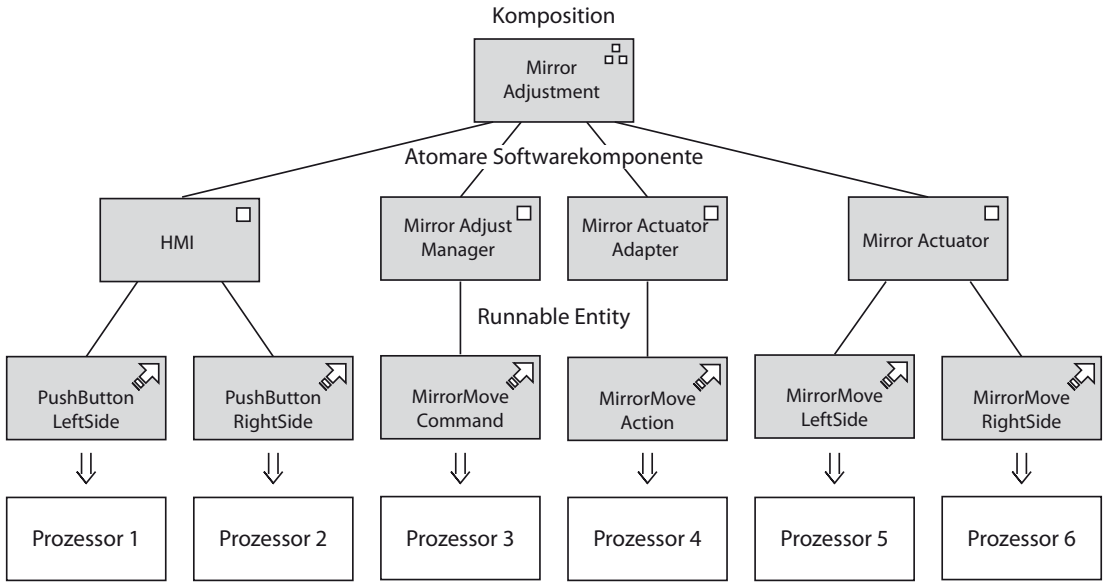


Abbildung 7.8: Prozessor-Mapping der Runnable Entities einer Softwarekomposition am Beispiel einer Spiegelverstellung[19]

Wie in der Abbildung 7.8 zu sehen ist, ergeben sich aus der Softwarestruktur von AUTOSAR drei verschiedene Ebenen für Funktionssoftware[20]: Kompositionen, atomare Softwarekomponenten und Runnable Entities. Die Design-Entscheidung, ob einzelne Runnables, atomare Softwarekomponenten oder ganze Kompositionen auf einem Prozessor bzw. Lokalspeicher effizient abgebildet werden können, hängt von der benötigten Rechenleistung, dem Speicherplatzbedarf und der Kommunikation zwischen SW-Cs bzw. Runnables ab. Aus dieser Entscheidung resultiert die Anzahl der benötigten Softprozessoren und damit der Ressourcenverbrauch des MPSoCs auf dem FPGA.

[18] vgl. 3.3.3 Parallele Datenverarbeitung, S. 42 f.

[19] Darstellung abgeleitet aus [AS09e, S. 16 f.]

[20] vgl. 7.2.2.2 Softwarestruktur, S. 205

7.2.3 Run-Time Environment (RTE)

Schnittstelle zwischen BSW und SW-C. Die *Run-Time Environment* (RTE) stellt im AUTOSAR-Schichtenmodell die Verbindung der *hardwareunabhängigen* Applikationssoftware (SW-C) zur *hardwareabhängigen* Basic Software (BSW) und den dort angebotenen Diensten, wie z.B. Kommunikations- und Speicherdienste, dar[21]. In Kooperation mit den Kommunikationsdiensten der Basic Software (COM) wird durch die RTE das Konzept des *Virtual Functional Bus*[22] (VFB) realisiert.

Application Interfaces. Die gebräuchlichsten Schnittstellen der RTE zur obersten Schicht, der Applikationssoftware, sind in der Spezifikation von AUTOSAR nach Domänen geordnet standardisiert worden[23].

Dynamische Erzeugung. Die Laufzeitumgebung RTE ist kein statisches Objekt, sondern wird, wie in der **Abbildung 7.9** dargestellt, anhand der extrahierten Daten aus der vorher definierten Konfiguration der ECU-Hardware und den zu implementierenden Softwarekomponenten für jedes Steuergerät dynamisch generiert. Die Rezentralisierung der Steuergeräte mittels Space-Sharing erfordert dagegen ein Software-First-Design[24], bei dem die Funktionssoftware den Ausgangspunkt bildet und das daher von dem in der AUTOSAR-Methodik spezifizierten Entwicklungsprozess abweicht. Aus diesem Grund muss der Ablauf in der Prozesskette von AUTOSAR für die Einführung von Space-Sharing neu geordnet werden[25].

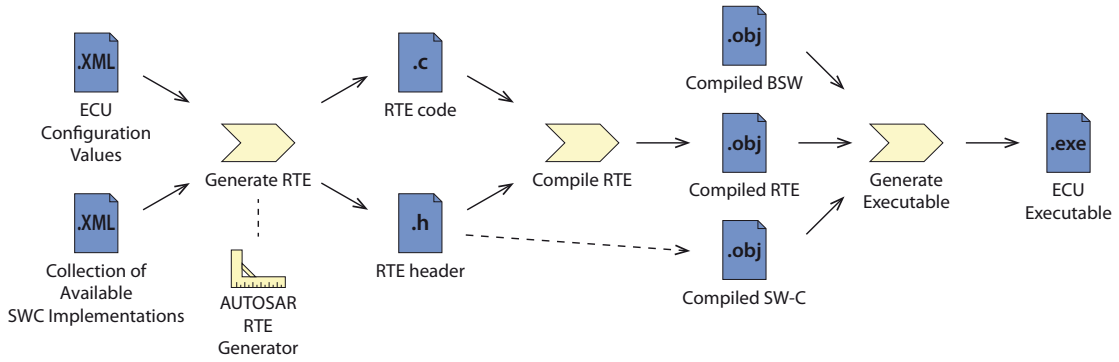


Abbildung 7.9: RTE Entwicklungsprozess[26]

[21] vgl. Abbildung 7.2: AUTOSAR Schichtenmodell, S. 201 ff.

[22] vgl. 7.3.1 ECU-Mapping, S. 209 f.

[23] vgl. [AS09f]

[24] vgl. 4.5.1 System-Design, S. 60 ff.

[25] vgl. 7.5 AUTOSAR-Methodik, S. 217 ff.

[26] vereinfachte Darstellung nach AUTOSAR System Build Methodology [AS09a, S. 53]

7.3 Kommunikation

Inhalt. Die Kommunikation zwischen den Steuergeräten und den darin implementierten Softwarekomponenten spielt aufgrund des hohen Vernetzungsgrades im Fahrzeug und nicht zuletzt aufgrund der Modularisierung der Anwendungssoftware durch AUTOSAR eine ganz entscheidende Rolle bei der Rezentralisierung der Steuergeräte durch Space-Sharing. In diesem Kapitel geht es darum, wie die Kommunikation zwischen verschiedenen Softwarekomponenten in AUTOSAR spezifiziert ist und wie dies bei der Rezentralisierung im Echtzeitparallelrechner bzw. MPSoC abgebildet werden kann.

7.3.1 ECU-Mapping

Kommunikation zwischen SW-Cs. Eine erlebbare Funktion im Fahrzeug wird in AUTOSAR i.d.R. mit mehreren Softwarekomponenten (SW-Cs) realisiert, wobei jede SW-C eine bestimmte Teilfunktion übernimmt und in diesem Rahmen mit anderen SW-Cs interagiert bzw. kommuniziert. Die **Abbildung 7.10** zeigt als Beispiel die beteiligten SW-Cs bei einer manuellen Spiegelverstellung von der Bedieneinheit (HMI) bis zum Stellmotor (Mirror Actuator) und deren Kommunikationsverbindungen.

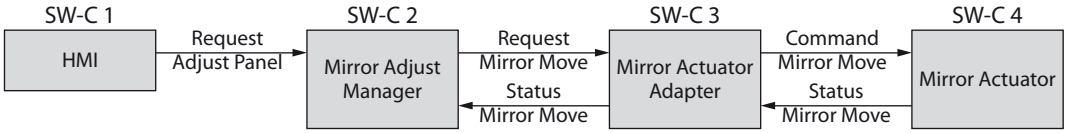


Abbildung 7.10: Softwarekomponenten zur manuellen Spiegelverstellung[27]

Verteiltes System. Die miteinander agierenden Softwarekomponenten können im verteilten System je nach Systemarchitektur auf ein einzelnes Steuergerät oder auf mehrere Steuergeräte gemappt sein. Dementsprechend erfolgt die Kommunikation zwischen den Softwarekomponenten steuergeräteintern oder über externe Busverbindungen zwischen den Steuergeräten (**Abbildung 7.11**). Bei der Rezentralisierung der Steuergeräte mittels Space-Sharing erfolgt diese Kommunikation nur innerhalb des Echtzeitparallelrechners, so dass die externen Busverbindungen zwischen den Steuergeräten durch NoC-Verbindungen zwischen den Prozessoren im MPSoC ersetzt werden.

[27] vereinfachte Darstellung nach [AS09e, S. 17]

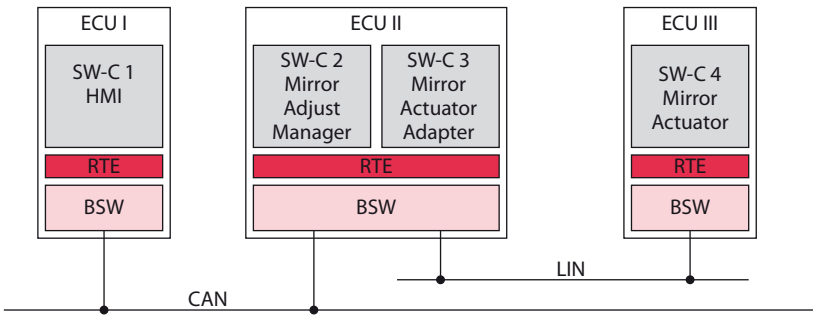


Abbildung 7.11: Klassisches Mapping von Softwarekomponenten auf verschiedene ECUs am Beispiel einer Spiegelverstellung[28]

7.3.2 Virtual Functional Bus (VFB)

Abstraktion auf Systemebene. Die Interaktion zwischen den Softwarekomponenten (SW-Cs) erfolgt über deren Port Interfaces. Dabei spielt es oberhalb der RTE auf der Ebene der Funktionssoftware keine Rolle, ob Kommunikationsverbindungen steuergeräteintern oder steuergeräteübergreifend aufgebaut sind. In AUTOSAR werden die Kommunikationsverbindungen auf der Anwendungsebene durch den *Virtual Functional Bus* (VFB) abstrahiert (**Abbildung 7.12**).

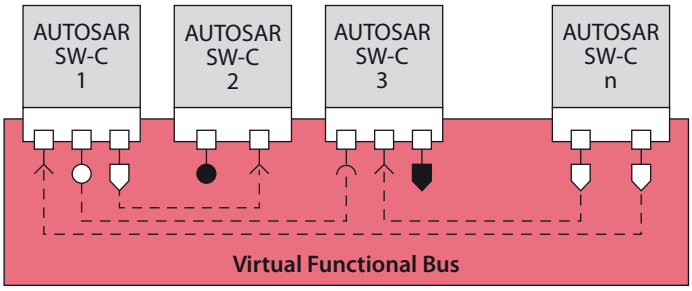


Abbildung 7.12: Kommunikation auf VFB-Ebene[29]

Transparente Kommunikationswege. Für SW-Cs bedeutet das VFB-Konzept, dass die Kommunikationswege für die AUTOSAR Funktionssoftware transparent gehalten werden. Wie die Kommunikation im realen System erfolgt, ist letztlich Aufgabe der RTE und der darunterliegenden Basis Software, wobei die Kommunikationswege spätestens zum Zeitpunkt der Generierung der RTE bekannt sein müssen. Da die RTE aus Sicht der SW-Cs für die interne und externe Kommunikation gleichermaßen zuständig ist, bleibt die Funktionssoftware unabhängig von

[28] grafische Darstellung nach [AS08c, S. 9]

[29] grafische Darstellung aus [AS09d, S. 9]

der Systemkonfiguration und ist somit auch auf andere Steuergeräte portierbar. Für die Rezentralisierung von Steuergeräten durch *ConPar* bedeutet dies wiederum, dass die Kommunikation zwischen AUTOSAR-konformen SW-Cs auf VFB-Ebene durch die Interprozessorkommunikation im MPSoC abgebildet werden kann, da es im VFB-Konzept keine Rolle spielt, ob die Kommunikation geräteintern oder -extern erfolgt. Es müssen lediglich die Port Interfaces der Funktionssoftware entsprechend ihrer Konfiguration bedient werden.

Kommunikationsmodelle. Die Laufzeitumgebung RTE stellt zwei Modelle für den Datenaustausch zwischen Softwarekomponenten zur Verfügung: *Sender-Receiver-Kommunikation* (signal passing) und *Client-Server-Kommunikation* (function invocation). Beide Kommunikationsmodelle funktionieren dem VFB-Konzept entsprechend zwischen Applikationen innerhalb einer ECU als auch zwischen Applikationen auf getrennten ECUs [Ki09, S. 100].

7.3.3 Sender-Receiver-Kommunikation

Senden und Empfangen von Daten. Die Sender-Receiver-Kommunikation in AUTOSAR beinhaltet das unidirektionale Senden und Empfangen von Signalen. Ein Sender-Receiver-Interface kann ein oder mehrere Datenelemente enthalten, welche selbst einfache oder komplexe Datentypen beinhalten. Für eine zeitgleiche Übertragung können mehrere Datenelemente in einer Struktur gekaspelt werden [Ki09, S. 101]. Der Sender erzeugt eine Instanz dieser Datenstruktur mit den benötigten Daten bevor die RTE zum Senden der Daten aufgefordert wird.

Explizite und implizite Kommunikation. Die RTE unterstützt zwei Formen der Sender-Receiver-Kommunikation: die explizite und die implizite Kommunikation [AS09a, S. 48]. Eine vergleichende Übersicht zwischen impliziter und expliziter Kommunikation befindet sich in [AS09a, S. 195]. Beide Formen sollen nun nachfolgend näher betrachtet werden.

7.3.3.1 Implizite Kommunikation

Implizites Lesen von Daten. Für das implizite Lesen von Daten (*DataReadAccess*) werden die Daten beim Start der Runnable durch einen Kopiervorgang zur Verfügung gestellt und nicht mehr verändert bis die Runnable terminiert. Wenn eine Runnable startet, liest die RTE die Daten und stellt der Runnable eine Kopie

der Daten zur Verfügung. So bedeutet ein *IMPLICIT_RECEIVE*, dass die Runnable auf einen zum Startzeitpunkt aktualisierten Wert zugreift [AS09d, S. 41]. Diese Vorgehensweise setzt allerdings voraus, dass die Runnable auch irgendwann terminiert. DataReadAccess ist nur für Datenelemente mit Datensemantik (*isQueued* = *false*) erlaubt.

Implizites Schreiben von Daten. Beim impliziten Schreiben von Daten (*DataWriteAccess*) werden die geschriebenen Daten der Runnable erst gesendet, nachdem die Runnable terminiert ist. Ein *IMPLICIT_SEND* bedeutet, dass ein Wert zur Laufzeit modifiziert, aber erst nach der Terminierung der Runnable gesendet wird [AS09d, S. 39]. Das Senden der Daten geschieht somit unabhängig vom Zeitpunkt der Verarbeitung in der Runnable. Bei multiplen Schreibzugriffen ist das letzte implizit geschriebene Datum auch das gesendete Datum (last-is-best-Semantik). Auch hier gilt, dass die Runnable eine endliche Ausführungszeit besitzen müssen.

Datenkonsistenz zur Laufzeit. Der Vorteil dieser Methode besteht in der Datenkonsistenz zur Laufzeit der Runnable. Mit dem Start und dem Ende der Laufzeit existieren zwei definierte Zeitpunkte zur Aktualisierung von externen Daten[30]. Eine äquivalente Methode wird auch bei speicherprogrammierbaren Steuerungen eingesetzt, wo das Prozessabbild der Eingänge am Zyklusbeginn gelesen und das Prozessabbild der Ausgänge am Zyklusende geschrieben wird.

7.3.3.2 Explizite Kommunikation

Explizites Lesen von Daten. Bei einer expliziten Kommunikation ist das Runnable für die Aktualisierung der externen Daten zuständig. Hierfür werden dem Runnable entsprechende Funktionsaufrufe für das Senden und Empfangen externer Daten durch die RTE bereitgestellt. Das Verhalten expliziter Zugriffe zum Lesen von Werten kann *nicht-blockierend* oder *blockierend* sein.

Nichtblockierende Kommunikation. Erfolgt der Aufruf nicht-blockierend, so wird sofort ein Wert zurückgegeben. Steht kein aktueller Wert zur Verfügung, so wird der zuletzt empfangene Wert zurückgegeben (*isQueued* = *false*) oder angezeigt, dass kein aktueller Wert (*isQueued* = *true*) zur Verfügung steht [AS09a, S.193].

[30] vgl. Kommunikation von Echtzeitdaten durch periodisches Scheduling von Nachrichten in 5.4.3 Nachrichten-Scheduling, S. 92 ff.

Blockierende Kommunikation. Bei einem blockierendem Kommunikationsaufruf wird die Programmabarbeitung des Runnables unterbrochen, bis neue Daten vorhanden sind. Sobald neue Daten anliegen, setzt die RTE die Ausführung der wartenden Runnable fort. Um ein unendliches Warten zu vermeiden, kann für einen blockierenden Aufruf ein Timeout definiert werden [AS09a, S.193].

Explizites Schreiben von Daten. Das explizite Schreiben von Daten ist immer nicht blockierend. Trotzdem kann ein Runnable der Kategorie 2 auf die Bestätigung des erfolgreichen Schreibversuchs warten (*AcknowledgementRequest*).

7.3.4 Client-Server-Kommunikation

Bereitstellen und Nutzen von Diensten. Die Client-Server-Kommunikation ist eine bidirektionale Kommunikation zwischen einem Client, welcher einen angebotenen Dienst nutzt und einem Server, der diesen Dienst anbietet [Ki09, S. 105]. Eine Anfrage (*Request*) wird vom Client an den Server gesendet und eine Antwort (*Response*) geht vom Server zurück an den Client. Der Client initiiert die Kommunikation, indem er einen Dienst vom Server anfordert und eventuell benötigte Parameter übergibt [AS09a, S. 215]. Der Aufruf kann entweder synchron (*wait for server to complete*) oder asynchron erfolgen. Ein sehr übersichtlicher Vergleich zwischen synchroner und asynchroner Client-Server-Kommunikation befindet sich in [AS09a, S. 228 f.].

7.3.4.1 Synchrone Kommunikation

Blockierende Kommunikation. Bei einem synchronen Aufruf blockiert die Client-Runnable, bis die Antwort der Server-Runnable empfangen ist. Für den Fall, dass keine Antwort eintrifft, existiert eine Timeout-Überwachung seitens der RTE.

7.3.4.2 Asynchrone Kommunikation

Nichtblockierende Kommunikation. Bei einem asynchronen Aufruf steht der Timeout in der Ergebnisfunktion (*Rte_Result*). Ein erneuter Aufruf bei einer noch ausstehenden Antwort des Servers liefert ein *RTE_E_LIMIT*.

7.3.5 Multiplizität

Mehrere Sender bzw. Empfänger. Die RTE unterstützt neben der Punkt-zu-Punkt Kommunikation auch die Kommunikation mit mehreren Sendern bzw. Empfängern. Mehrere Empfänger (*1:n*) werden allerdings nur bei der Sender-Receiver-Kommunikation unterstützt. Dagegen werden mehrere Sender (*n:1*) sowohl bei der Sender-Receiver-Kommunikation als auch bei der Client-Server-Kommunikation unterstützt [AS09b, S. 42-43].

7.3.6 Interrunnable-Variable

SW-C-interne Kommunikation. Die Sender-Receiver Kommunikation, wie auch die Client-Server Kommunikation, erfolgt über das Port-Interface einer Softwarekomponente (SW-C). Die kleinste strukturelle Einheit stellt jedoch das Runnable[31] dar, wobei eine Softwarekomponente mehrere Runnables enthalten darf. Für die Kommunikation zwischen Runnables innerhalb einer SW-C beschreibt die AUTOSAR Spezifikation einen weiteren Mechanismus, die *Interrunnable-Variable*[32]. Die Kommunikation mittels Interrunnable-Variablen basiert auf dem Prinzip des gemeinsamen Speichers. Um die Portierbarkeit der SW-C zu erhalten, können nur Runnables innerhalb einer SW-C darauf zugreifen, wodurch gleichzeitig eine Kapselung SW-C-interner Daten erfolgt. Der Datenaustausch zwischen Runnables verschiedener SW-Cs muss deshalb zwingend über das Port-Interface der SW-C erfolgen.

Interrunnable-Variablen bei ConPar. Aufgrund der Kapselung von Speicher[33] und Prozessoren[34] durch Space-Sharing kann die Kommunikation durch Interrunnable-Variablen bei ConPar nicht prozessorübergreifend erfolgen. Da diese Kommunikationsform jedoch nur auf eine einzelne SW-C begrenzt ist, gilt für Space-Sharing lediglich die Einschränkung, dass die miteinander kommunizierenden Runnables der SW-C eine atomare Einheit darstellen und nicht auf verschiedene Prozessoren verteilt werden dürfen. Eine Alternative bietet die Implementierung eines exklusiven gemeinsamen Speichers für Interrunnable-Variablen, bei Xilinx FPGAs z.B. in Form einer Mailbox, zwischen zwei Prozessoren.

[31] vgl. 7.3 Kommunikation, S. 209 ff.

[32] vgl. [AS09a, S. 230]

[33] vgl. 4.4.2 Räumliche Isolation durch Space-Sharing, S. 59 f.

[34] vgl. 4.3.2 Zeitliche Isolation durch Space-Sharing, S. 56 ff.

Sender-Receiver Verhalten. Aus Gründen der Datenkonsistenz erfolgt der Zugriff auf Interrunnable-Variablen nicht direkt, sondern über die Zugriffsfunktionen der RTE[35], weshalb sich deren Kommunikationsaufruf analog zur Sender-Receiver Kommunikation verhält [AS09a, S. 230]. Auch die Unterscheidung zwischen explizitem und implizitem Verhalten bleibt erhalten. Der einzige Unterschied liegt in der Datenkapselung und -transparenz begründet. Vor diesem Hintergrund wäre auch eine Realisierung wie bei einer SW-C übergreifenden Sender-Receiver Kommunikation möglich, wodurch das Runnable als kleinste strukturelle Größe bei der Zuordnung von Prozessoren im MPSoC verbleibt.

7.4 Rezentralisierung von Steuergeräten

Inhalt. Dieser Abschnitt behandelt die Rezentralisierung von Steuergeräten, welche die grundlegende Motivation zur Entwicklung des Echtzeitparallelrechners *ConPar* darstellt. Auf Basis der vorangegangenen Abschnitte dieses Kapitels wird gezeigt, wie die Funktionssoftware verschiedener Steuergeräte in einem Multiprozessorsystem konzentriert werden kann, um auf diese Weise der Forderung nach Reduzierung der Anzahl der Steuergeräte im Fahrzeug[36] zu entsprechen.

7.4.1 Portierung der Steuergerätesoftware

Portierbarkeit von Funktionssoftware. Mit der Einführung des Schichtenmodells in AUTOSAR[37] wird die Funktionalität von Steuergeräten in Form von SW-Cs für andere (Hardware-)Plattformen portierbar. Dies erlaubt wiederum die Migration von Funktionssoftware auf den Echtzeitparallelrechner *ConPar*.

7.4.2 Prozessor-Mapping

Prozessor-Mapping. Wie die **Abbildung 7.13** zeigt, werden die Softwarekomponenten (SW-Cs) auf die Prozessoren des MPSoC gemappt. Entscheidend für die Anzahl der Prozessoren ist die Granularität der Funktionssoftware[38], da nur nebenläufige Teile auf verschiedene Prozessoren gemappt werden können.

[35] vgl. [Ki09, S. 109]

[36] vgl. 2.2 Pro und Kontra Rezentralisierung, S. 11 ff.

[37] vgl. Abbildung 7.1: Abstraktion der Hardware in AUTOSAR, S. 200

[38] vgl. 7.2.2.3 Prozessor-Mapping, S. 206 f.

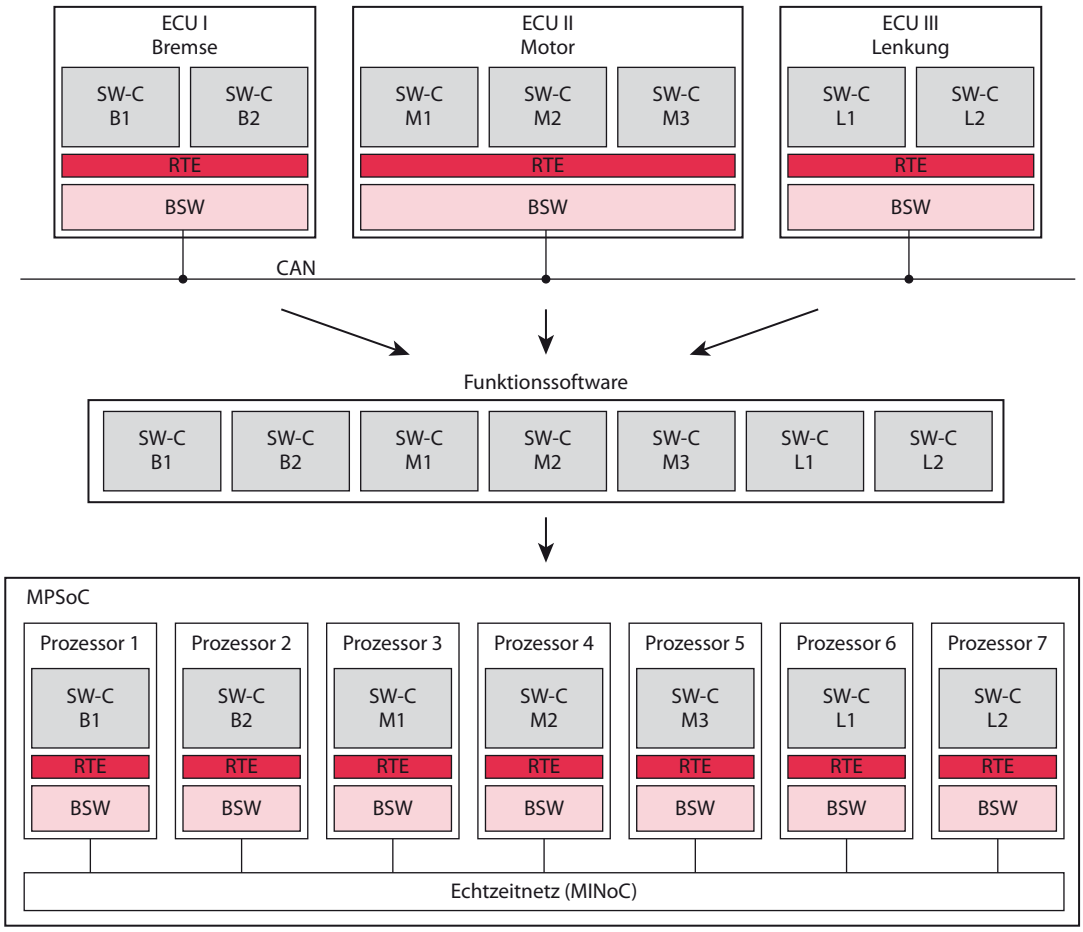


Abbildung 7.13: Rezentralisierung der Funktionssoftware (SW-C) von verschiedenen Steuergeräten (ECU) in einem Multiprozessorsystem on-Chip (MPSoC)

7.4.3 Kommunikation

Kommunikation im verteilten System. In der aktuell existierenden dezentralen Bordnetzarchitektur[39] sind die Steuergeräte im Fahrzeug als verteiltes System über verschiedene Bussysteme wie CAN oder FlexRay[40] miteinander verbunden. Die Kommunikation zwischen den Steuergeräten ist über den Austausch von Nachrichten auf dem jeweiligen Bus realisiert, wobei der Nachrichtenaustausch aus Gründen der Vorhersagbarkeit i.d.R. zyklisch und mit fester Periodizität erfolgt. Dieses Vorgehen erlaubt eine gute Aussage bezüglich der Kommunikationslast auf dem Bus und damit über die zu erwartende Latenz in der Nachrichtenübertragung. Das Grundproblem der Kommunikation in Bussystemen, der schlechten Skalierbarkeit, kann angesichts der steigenden Komplexität in der Automobilelektronik

[39] vgl. Abbildung 2.1: Dezentrale Bordnetzarchitektur, S. 10

[40] vgl. 2.2.1 Kommunikation, S. 11 ff.

jedoch nicht gelöst werden. So werden aufgrund der gestiegenen Kommunikationslast im Audi A8 der aktuellen Generation neun verschiedene Bussysteme zum steuergeräteübergreifenden Nachrichtenaustausch verwendet, wie die **Tabelle 7.1** zeigt. Eine grafische Übersicht über alle Steuergeräte und Bussysteme im Audi A8 befindet sich in [Au09a, S. 10-11].

Tabelle 7.2: Verwendete Bussysteme im Audi A8[41]

Bussystem	Datenübertragung
CAN-Antrieb	500 kbit/s
CAN-Komfort	500 kbit/s
CAN-Extended	500 kbit/s
CAN-Anzeige und Bedienung	500 kbit/s
CAN-Diagnose	500 kbit/s
FlexRay	10 Mbit/s
MOST-Bus	22,5 Mbit/s
LIN-Bus	20 kbit/s
Sub-Bus-System	500 kbit/s

Kommunikation im Echtzeitparallelrechner. In einer rezentralisierten Bordnetzarchitektur befindet sich die gesamte Funktionssoftware des Fahrzeugs in einem zentralen „Steuergerät". Auf diese Weise wird die steuergeräteübergreifende Kommunikation der Funktionssoftware auf eine steuergeräteinterne Interprozessorkommunikation im Echtzeitparallelrechner verlagert. Die nachrichtenbasierte Kommunikation kann aus dem verteilten System übernommen werden, wobei die Verbindung zwischen den Softwarekomponenten nicht über einen Bus, sondern über ein echtzeitfähiges Verbindungsnetzwerk[42] realisiert ist (Abbildung 7.13).

7.5 AUTOSAR-Methodik

Inhalt. In diesem Abschnitt wird die Vorgehensweise bei der Entwicklung von Steuergerätesoftware nach der in AUTOSAR spezifizierten Methodik betrachtet. Der besondere Fokus liegt nachfolgend in den Gemeinsamkeiten und den Unterschieden zwischen dem Ansatz der bereits existierenden AUTOSAR-Methodik und dem Software-First-Design im FPGA-basierten Space-Sharing und wie letzteres in AUTOSAR migriert werden könnte.

[41] aus [Au09a, S. 12]

[42] vgl. Kapitel 5 - Interprozessorkommunikation, S. 73 ff.

7.5.1 Grundlagen

7.5.1.1 Arbeitsablauf

Der Weg zum Steuergerät. Eine der herausragenden Veränderungen bei der Einführung des AUTOSAR-Standards ist die Möglichkeit der toolgestützten Automatisierung von Arbeitsabläufen. Mit AUTOSAR existiert eine Spezifikation, welche den Arbeitsproduktfluss innerhalb des Entwicklungsprozesses, von der Spezifikation bis zum funktionierenden Steuergerät, beschreibt. AUTOSAR ermöglicht damit eine einheitliche Vorgehensweise bei der Entwicklung von Steuergerätesoftware sowie den Austausch von standardisierten Arbeitsprodukten zwischen den verschiedenen Tools innerhalb der Werkzeugkette. Der Weg zum fertigen Steuergerät wird in AUTOSAR unter dem Begriff der AUTOSAR-Methodik[43] beschrieben. In der AUTOSAR-Methodik sind wesentliche Abhängigkeiten, Aktivitäten und Arbeitsprodukte bei der Steuergeräteentwicklung, von der Konfiguration auf Systemebene bis zur Erzeugung ausführbaren Programmcodes auf dem Steuergerät, festgehalten. Während AUTOSAR in der Version 3.1 explizit keine Rollen und Verantwortlichkeiten festlegte[44], so wurde die AUTOSAR-Methodik in der Version 4.0 um eben diese Eigenschaften erweitert[45], welche nun die Zuständigkeiten bei Arbeitsprodukten und Aktivitäten im Entwicklungsprozess definieren.

7.5.1.2 Grafische Notation

Unified Modeling Language. Die Spezifikation von AUTOSAR verwendet in der grafischen Darstellung der AUTOSAR-Methodik das *Software Process Engineering Metamodel* (SPEM) der Object Management Group [OM08]. SPEM ist ein UML-Profil, welches die Integration der AUTOSAR-Methodik in das Meta-Modell von AUTOSAR ermöglicht. Die AUTOSAR-Methodik verwendet für die Darstellung der Arbeitsabläufe nur wenige Elemente von SPEM, deren grafische Notation in [AS08b, S. 7-11] beschrieben ist.

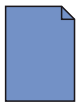
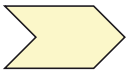
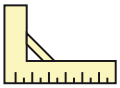
[43] engl. AUTOSAR Methodology, beschrieben in [AS08b]

[44] aus „2.1.1 Scope of the methodology“ in [AS08b, S. 6]

[45] eine sehr umfangreiche Dokumentation der AUTOSAR-Methodik befindet sich in AUTOSAR Version 4.0 unter ../Methodology and Templates/Methodology/Auxiliary/Publish/index.html

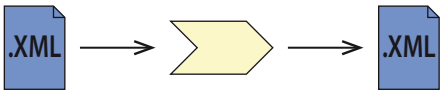
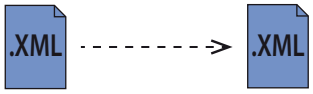
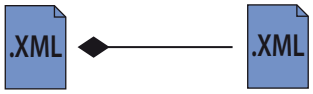
Grafische Elemente. Im Arbeitsproduktfluss der AUTOSAR-Methodik werden aus vorhandenen *Arbeitsprodukten* durch bestimmte *Aktivitäten* neue Arbeitsprodukte erzeugt. Die für Aktivitäten benötigten zusätzlichen Informationen und Werkzeuge werden in AUTOSAR als *Führungselemente* bezeichnet. Die **Tabelle 7.3** zeigt die in der AUTOSAR-Methodik verwendeten Symbole.

Tabelle 7.3: Grafische Elemente in AUTOSAR[46]

	*Arbeitsprodukt (work product)* ist eine Information oder ein Objekt, welches von einer Aktivität erzeugt bzw. verwendet wird.
	*Aktivität (activity)* beschreibt eine Tätigkeit, die von einer einzelnen Person oder einer Gruppe von Personen ausgeführt wird.
	*Führungselement (guidance)* beschreibt zusätzliche Informationen oder Werkzeuge, welche für die Ausführung einer Aktivität zur Verfügung stehen.

Darstellung von Beziehungen. Die AUTOSAR-Methodik beschreibt neben den einzelnen Arbeitsprodukten auch die Beziehungen der Arbeitsprodukte untereinander. Die **Tabelle 7.4** zeigt, wie die verschiedenen Beziehungen in der AUTOSAR-Methodik grafisch dargestellt werden.

Tabelle 7.4: Beziehungen in AUTOSAR[47]

	*Arbeitsproduktfluss (flow of work-products)* dient der Darstellung von Eingang und Ausgang von Aktivitäten und ist gekennzeichnet durch einen von der Quelle zum Ziel gerichteten Pfeil.
	*Abhängigkeit (dependency)* wird durch einen gestrichelten Pfeil dargestellt und zeigt an, dass ein Arbeitsprodukt von einem anderen Arbeitsprodukt abhängig ist.
	*Komposition (composition)* wird durch eine Linie mit einer ausgefüllten Raute am Ende dargestellt und zeigt an, dass ein Arbeitsprodukt andere Arbeitsprodukte beinhaltet.

[46] AUTOSAR Release 3.1 [AS08b, S. 8]

[47] AUTOSAR Release 3.1 [AS08b, S. 9 f.]

7.5.2 Steuergerätezentrierte Methodik

Das Steuergerät als zentrales Ziel. Die gegenwärtig existierende Topologie in der Automobilelektronik besteht aus einem verteilten System aus Steuergeräten[48]. Der Arbeitsproduktfluss der AUTOSAR-Methodik konzentriert sich demzufolge auch auf die Implementierung des AUTOSAR-Schichtenmodells[49] auf einzelne Steuergeräte und rückt damit das Steuergerät als Endprodukt in den Mittelpunkt der automobilen Softwareentwicklung. Die **Abbildung 7.14** fasst den in [AS08b, S. 18-32] beschriebenen Ablauf der AUTOSAR-Methodik in einer grafischen Übersicht zusammen. Eine vollständige grafische Darstellung des Produktflusses befindet sich in [AS08b, S. 37].

Ausgangspunkt. Die Softwareentwicklung unter AUTOSAR ist auf die Generierung von Steuergerätesoftware ausgerichtet, wobei neben der Steuergerätehardware auch Funktionalität des Steuergerätes in Form der Funktionssoftware (SW-C) fest definiert ist. Den Ausgangspunkt der AUTOSAR-Methodik bilden aus diesem Grund die Beschreibungen der Systemkonfiguration, d.h. der Hardware des Steuergerätes und des umgebenden Systems im Fahrzeug, und der als SW-Cs verfügbaren Funktionssoftware (Abbildung 7.14 a).

Generierung der Basis-Software. Aus den Beschreibungen der zugrundeliegenden Hardware des Steeuergerätes, dem umgebenden System und der zu implementierenden Funktionssoftware wird die AUTOSAR Basis-Software konfiguriert. Hierzu zählen alle Ebenen des AUTOSAR-Schichtenmodells zwischen den SW-Cs und der Hardware. Dazu gehören u.a. das Betriebssystem, Speicherdienste, Kommunikationsdienste bis zur Laufzeitumgebung RTE. Nach der Kompilierung aller Komponenten entsteht als Ergebnis des Arbeitsproduktflusses eine auf dem Steuergerät ausführbare Datei, welche die Funktions- und Basis-Software des Steuergerätes beinhaltet (Abbildung 7.14 b).

[48] vgl. 2.1.1 Dezentrale Topologie, S. 9 f.

[49] vgl. Abbildung 7.2: AUTOSAR Schichtenmodell, S. 201

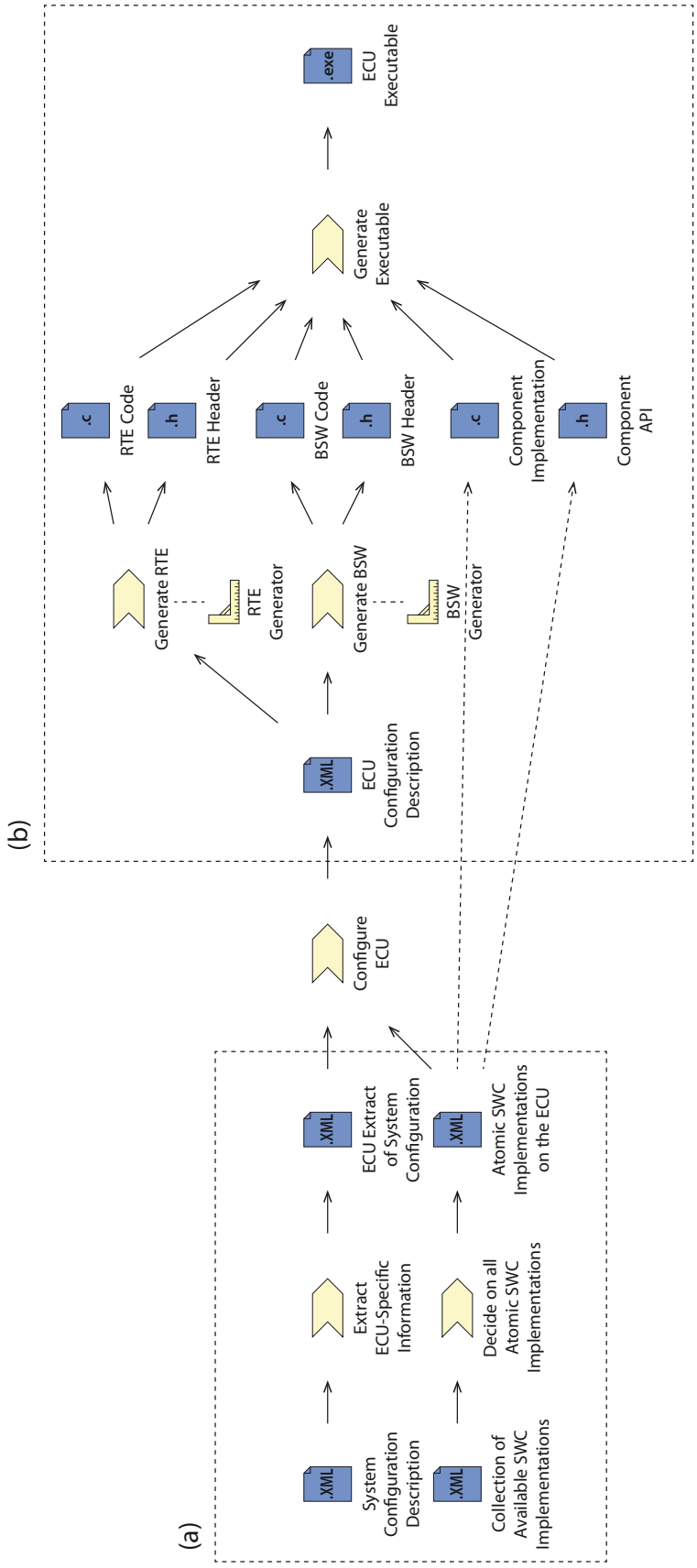


Abbildung 7.14: Steuergeräte-Entwicklung nach AUTOSAR

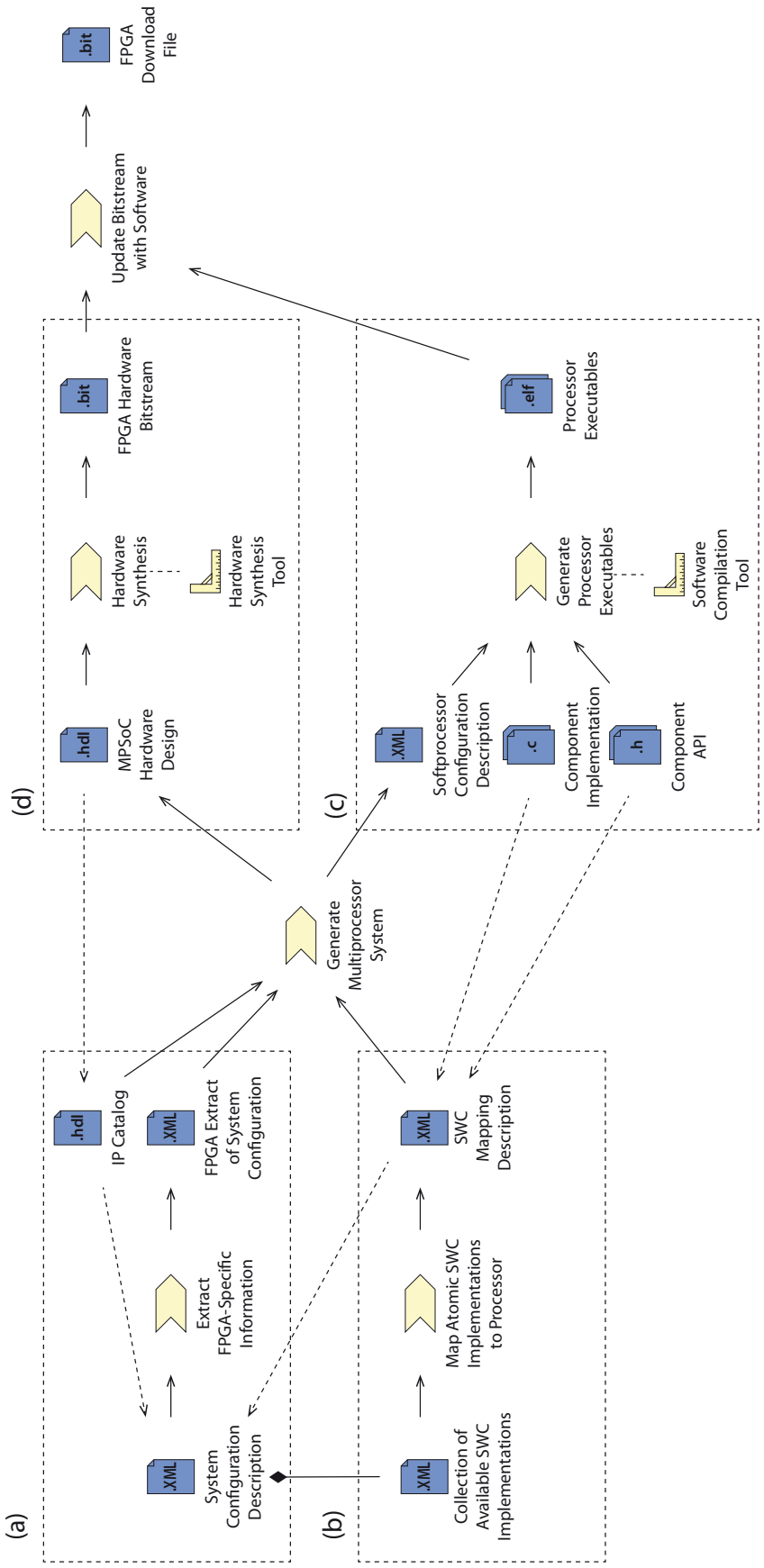


Abbildung 7.15: Steuergeräte-Entwicklung mit Space-Sharing

7.5.3 Funktionsorientierte Methodik

Die Funktionalität als zentrales Ziel. Bei der funktionsorientierten Methodik geht es darum, die Funktionalität des Fahrzeuges zu realisieren. Der Produktfluss ist in der **Abbildung 7.15** dargestellt. Hierbei steht zunächst die Funktionssoftware im Mittelpunkt, für die eine geeignete Rechenhardware auf FPGA-Basis erstellt wird.

Generierung von Hard- und Software. Wie bei der AUTOSAR-Methodik bilden auch bei der funktionsorientierten Methodik die Beschreibung der Systemkonfiguration (Abbildung 7.15 a) und die Beschreibung der Funktionssoftware (Abbildung 7.15 b) den Ausgangspunkt. Während sich die AUTOSAR-Methodik jedoch auf die Generierung der Steuergerätesoftware beschränkt, teilt sich der Produktfluss in der funktionsorientierten Methodik in den bekannten Softwareteil (Abbildung 7.15 c) und einen zusätzlichen Hardwareteil (Abbildung 7.15 d). Im Hardwareteil wird aus der Beschreibung der Systemkonfiguration und der Beschreibung der Funktionssoftware eine geeignete Multiprozessorumgebung auf FPGA-Basis generiert[50]. Auf diese Weise wird im FPGA-basierten Multiprozessorsystem die Rechnerarchitektur Teil des Software-Engineering-Prozesses. Der Softwareteil unterscheidet dagegen sich nur unwesentlich von der AUTOSAR-Methodik. Nach dem Mapping der SW-Cs auf Prozessoren (Abbildung 7.15 b) wird für jeden Prozessor im System eine ausführbare Datei erzeugt (Abbildung 7.15 d).

7.6 Fazit

Softwarearchitektur. Die Spezifikation von AUTOSAR beschreibt ein Software-Schichtenmodell für automobile Steuergeräte, welches sich mehr und mehr als zukünftiger Standard für die Steuergeräteentwicklung etabliert.

Portierbarkeit. Die Einführung des Schichtenmodells von AUTOSAR führt zu einer strikten Trennung der Steuergerätesoftware in Funktions- und Basissoftware. In Verbindung mit der Standardisierung von Schnittstellen zwischen der Laufzeitumgebung RTE und der Funktionssoftware SW-C erlaubt AUTOSAR die Portierung der Funktionalität von Steuergeräten auf verschiedene andere Hardware-Plattformen. Dies bedeutet wiederum, dass mit AUTOSAR auch eine Portierung der Steuergerätesoftware auf den Echtzeitparallelrechner *ConPar* und damit eine Rezentralisierung der bestehenden Steuergerätesoftware möglich ist.

[50] vgl. 4.2.1 Space-Sharing mit FPGAs, S. 51

Kommunikation. Da sowohl die Kommunikation zwischen den Steuergeräten, als auch die Interprozessorkommunikation im Echtzeitparallelrechner auf dem Austausch von Nachrichten basiert, kann die bestehende Kommunikation zwischen SW-Cs unterschiedlicher Steuergeräte ohne Einschränkungen auf eine Interprozessorkommunikation im Echtzeitparallelrechner *ConPar* übertragen werden.

AUTOSAR-Methodik. Neben der Softwarearchitektur beschreibt AUTOSAR auch eine Methodik für die Entwicklung von Steuergerätesoftware. Im Wesentlichen handelt es sich dabei um die Generierung der Basis-Software, wobei neben der Funktionssoftware auch die Steuergeräte-Hardware als fester Ausgangspunkt dient. Im Zuge von Space-Sharing und dem damit verbundenen Software-First-Design muss der Produktfluss in der AUTOSAR-Methodik für *ConPar* angepasst werden, so dass neben der Basis-Software auch die FPGA-basierte Rechenhardware generiert wird.

Kapitel 8 - Zusammenfassung

Ziel der Arbeit. Die vorliegende Arbeit befasst sich mit dem Entwurf eines Echtzeitparallelrechners vor dem Hintergrund der Rezentralisierung von Steuergeräten im Automobil. Hierfür wurden die besonderen Eigenschaften der Echtzeit- und der Parallelrechner in einem Echtzeitparallelrechner vereint und in den bestehenden Standard der Automobilelektronik integriert.

Rezentralisierung von Steuergeräten. Die Idee für den Entwurf des Echtzeitparallelrechners basiert auf dem Konzept der Rezentralisierung von automobilen Steuergeräten auf einen oder wenigen zentralen Rechenknoten im Fahrzeug, verbunden mit einem homogenen Kommunikationssystem. Das **Kapitel 2** beschäftigt sich zunächst mit der Frage, welche Vorteile oder Nachteile das Konzept der Rezentralisierung von Steuergeräten gegenüber der gegenwärtigen E/E-Architektur besitzt und wie die anstehenden Probleme der wachsenden funktionalen Komplexität und dem hohen Vernetzungsgrad der Steuergeräte durch Rezentralisierung gelöst werden können. So wurde gezeigt, dass das Konzept der zentralen Rechenknoten signifikante Vorteile bei der Skalierbarkeit, der Konfigurierbarkeit, der Systemintegration und der Portierbarkeit von Software bietet.

Echtzeitparallelrechner. Der Echtzeitparallelrechner muss in der Lage sein, ein von Fahrzeugtyp und -ausstattung abhängiges Volumen an zeitkritischen Daten zu verarbeiten. Das heißt, der Rechner muss eine hohe skalierbare Rechenleistung bereitstellen und gleichzeitig ein definiertes Zeitverhalten garantieren. Unter dieser Forderung treffen zwei bisher absolut eigenständige Rechnerwelten zusammen - die *Parallelrechner* und die *Echtzeitrechner*. Parallele Rechnerarchitekturen, wie sie auch im Bereich raumfüllender Höchstleistungsrechner eingesetzt werden, bieten sehr hohe Rechenleistungen, sind aber nicht für Echtzeitanwendungen konzipiert. Auf der anderen Seite sind konventionelle Echtzeitrechner, deren Zeitverhalten üblicherweise durch eine geeignete Scheduling-Strategie bestimmt wird, nicht für sehr viele Tasks bzw. Prozessoren geeignet. Aus diesem Grund beinhaltet das **Kapitel 3** eine Analyse der besonderen Eigenschaften von Echtzeit- und Parallelrechnern. Als Ergebnis wurde das Konzept für den Echtzeitparallelrechner *ConPar* vorgestellt, welcher die beiden oben aufgeführten Forderungen nach Rechenleistung und Echtzeitverhalten erfüllt.

Space-Sharing. Zu den großen Herausforderungen bei der Verarbeitung von Echtzeitdaten zählt das Task-Scheduling, welches bei einer höheren Zahl an zeitkritischen Tasks schnell komplex und schwierig wird. Erhöht sich zudem aufgrund der geforderten Rechenleistung auch noch die Zahl der Prozessoren, so kommen konventionelle Systeme schnell an ihre Grenzen. Das **Kapitel 4** stellt deshalb für den Echtzeitparallelrechner das Konzept des *Space-Sharings* vor. In diesem Konzept wird jeder Task ein eigener Prozessor zur Verfügung gestellt, wodurch das konventionelle Task-Scheduling entfällt. Die Voraussetzung für Space-Sharing bildet neben der Rechnerarchitektur das ebenfalls in diesem Kapitel vorgestellte *Software-First-Design*, einem neuen Design-Paradigma für Multiprozessorsysteme, welches wiederum auf der FPGA-Technologie basiert.

Interprozessorkommunikation. Für Parallelrechner stellt das Verbindungsnetzwerk zwischen den Prozessoren eine notwendige Komponente dar, die wesentliche Eigenschaften wie z.B. die Skalierbarkeit des Parallelrechners definiert. Beim Echtzeitparallelrechner entscheidet die Interprozessorkommunikation zudem über das Zeitverhalten bei der Datenverarbeitung. Aus diesem Grund wurden in **Kapitel 5** verschiedene existierende Verbindungsnetzwerke auf ihre Eignung für den Einsatz im Echtzeitparallelrechner analysiert. Im Ergebnis wurden zwei mögliche Lösungen auf der Basis mehrstufiger dynamischer Netze vorgestellt, die beide in der Lage sind, zeitkritische Daten zu übertragen.

Energieeffizienz. Ein ganz wesentlicher Aspekt beim Entwurf von Rechensystemen ist deren Energieeffizienz. In **Kapitel 6** wurde theoretisch und durch experimentelle Untersuchungen gezeigt, wo die möglichen Einsparpotentiale im Konzept von Space-Sharing und dem Software-First-Design liegen. Auf dieser Basis wurden verschiedene technische Maßnahmen erarbeitet, mit denen der Energieverbrauch im Echtzeitparallelrechner optimiert werden kann.

Integration. Neben der Frage nach der Architektur eines Echtzeitparallelrechners steht auch die Frage im Raum, wie das Konzept der Rezentralisierung in die bestehende Welt der Automobilelektronik integriert werden kann. Dazu wurde in **Kapitel 7** die Spezifikation von AUTOSAR betrachtet, welche die Architektur von automobiler Software und auch deren Entwicklungsprozesse standardisiert. Auf der Basis dieser Analyse wurde gezeigt, wie der Echtzeitparallelrechner in die zukünftige Softwareentwicklung integriert werden kann.

Literatur

[Aj09] Ajima, Y.; Sumimoto, S.; Shimizu, T.: *Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers.* IEEE Computer, Vol. 42, Nr. 11, November 2009, S. 36-40.

[Al10] Altera Corporation: *Introducing Innovations at 28 nm to Move Beyond Moore's Law.* Altera White Paper WP-01125-1.1, Juli 2010.

[AS08a] AUTOSAR GbR: *Specification of Graphical Notation.* AUTOSAR Release 3.1, Dokument-Nr. 206, Version 1.0.6, Juni 2008.

[AS08b] AUTOSAR GbR: *AUTOSAR Methodology.* AUTOSAR Release 3.1, Dokument-Nr. 068, Version 1.2.2, August 2008.

[AS08c] AUTOSAR GbR: *Technical Overview.* Version 2.2.2, August 2008.

[AS09a] AUTOSAR GbR: *Specification of RTE.* AUTOSAR Release 4.0, Dokument-Nr. 084, Version 3.0.0, Dezember 2009.

[AS09b] AUTOSAR GbR: *Requirements on RTE Software.* AUTOSAR Release 4.0, Dokument-Nr. 083, Version 2.0.0, November 2009.

[AS09c] AUTOSAR GbR: *Glossary.* AUTOSAR Release 4.0, Dokument-Nr. 055, Version 2.2.0, Dezember 2009.

[AS09d] AUTOSAR GbR: *Virtual Functional Bus.* AUTOSAR Release 4.0, Dokument-Nr. 056, Version 2.0.0, November 2009.

[AS09e] AUTOSAR GbR: *Explanation of Application Interfaces of the Body and Comfort Domain.* AUTOSAR Release 4.0, Dokument-Nr. 268, Version 1.1.0, Dezember 2009.

[AS09f] AUTOSAR GbR: T*able of Application Interfaces.* AUTOSAR Release 4.0, Dokument-Nr. 241, Version 2.0.0, Dezember 2009.

[At08] Atienza, D.; Angiolini, F.; Murali, S.; Pullini, A.; Benini, L.; De Micheli, G.: *Network-on-Chip design and synthesis outlook.* Integration, the VLSI Journal, 41(2), Februar 2008.

[Au09a] Audi AG: *Audi A8 - Bordnetz und Vernetzung*. Selbststudienprogramm Nr. 459, November 2009.

[Au09b] Aust, S.; Richter, H.: *Parallel Computer Technology - A Solution for Automobiles? How car engineers can learn from parallel computing*. 3rd International Conference on Advanced Engineering Computing and Applications in Sciences; ADVCOMP 2009, Sliema, Oktober 2009, S. 148-152.

[Au10a] Aust, S.; Richter, H.: *Space Division of Processing Power for Feed Forward and Feed Back Control in Complex Production and Packaging Machinery*. World Automation Congress; WAC 2010, Kobe, September 2010, S. 1-6.

[Au10b] Aust, S.; Richter, H.: *Ein Echtzeitparallelrechner zur Rezentralisierung von Steuergeräten im Automobil*. erschienen in: Tschöke, H.; Krahl, J.; Munack, A. (Hrsg.): *Innovative Automobiltechnik II*. Expert Verlag, 2010, S. 70-88.

[Au10c] Aust, S.; Richter, H.: *Real-time Processor Interconnection Network for FPGA-based Multiprocessor System-on-Chip (MPSoC)*. 4th International Conference on Advanced Engineering Computing and Applications in Sciences; ADVCOMP 2010, Florenz, Oktober 2010, S. 47-52.

[Au10d] Audi AG: *Audi A8 ´10 - Fahrerassistenzsysteme*. Selbststudienprogramm Nr. 461, 2010.

[Au11a] Aust, S.; Richter, H.: *Skalierbare Rechensysteme für Echtzeitanwendungen*. erschienen in: Halang, W. A. (Hrsg.): *Herausforderungen durch Echtzeitbetrieb*. Springer Verlag, 2011, S. 111-120.

[Au11b] Aust, S.; Richter, H.: *Energy-Aware MPSoC with Space-Sharing for Real-Time Applications*. 5th International Conference on Advanced Engineering Computing and Applications in Sciences; ADVCOMP 2011, Lissabon, November 2011, S. 54-59.

[Au12] Aust, S.; Richter, H.: *Energy-aware MPSoC for Real-time Applications with Space-Sharing, Adaptive and Selective Clocking and Software-first Design*. International Journal on Advances in Software, Vol. 5, Nr. 3 & 4, 2012, S. 368-377.

[Ba97] Bartsch, H.-J.: *Taschenbuch mathematischer Formeln.* 17. Auflage, München; Wien, Carl Hanser Verlag, 1997.

[Ba06] Bauke, H., Mertens, S.: *Cluster Computing. Praktische Einführung in das Hochleistungsrechnen auf Linux-Clustern.* Berlin; Heidelberg, Springer Verlag, 2006.

[Be02] Benini, L.; De Micheli, G.: *Networks on chips: a new SoC paradigm.* IEEE Computer, Vol. 35, Nr. 1, Januar 2002, S. 70-78.

[Be06] Benini, L.; De Micheli, G.: *Networks on Chips: Technology and Tools.* San Francisco, Morgan Kaufmann, 2006.

[Be09] Benra, J. T.; Halang, W. A.: *Software-Entwicklung für Echtzeitsysteme.* Heidelberg; London; New York, Springer Verlag, 2009.

[Be10] Beutelspacher, A.: *Lineare Algebra.* Vieweg Verlag, Wiesbaden, 7. Auflage, 2010.

[Be62] Beneš, V. E.: *On Rearrangeable Three-Stage Connecting Networks*. Bell Systems Technical Journal, Vol. 41, Nr. 5, September 1962, S. 1481-1492.

[Bj06] Bjerregaard, T.; Mahadevan, S.: *A Survey of Research and Practices of Network-on-Chip.* ACM Computing Surveys (CSUR), Vol. 38, Nr. 1, 2006.

[Bl10] Black, D. C.; Donovan, J.; Bunton, B.; Keist, A.: *SystemC: From the Ground Up.* 2. Auflage, New York; Dordrecht; Heidelberg; London, Springer Verlag, 2010.

[Bö06] Böttcher, A.: *Rechneraufbau und Rechnerarchitektur.* Berlin; Heidelberg; New York, Springer Verlag, 2006.

[Bo10] Bone, T.: *Applying Timing Analysis to Vehicle Networking at Daimler Group Research and Advanced Engineering.* 4. Symtavision NewsConference, Braunschweig, September 2010.

[Br99] Bronstein, I. N.; Semendjajew, K. A.; Musiol, G.; Mühlig, H.: *Taschenbuch der Mathematik.* 4. Auflage, Frankfurt am Main; Thun, Verlag Harri Deutsch, 1999.

[Br10] Bruckmeier, R.: *Ethernet for Automotive Applications*. Freescale Technology Forum, Orlando, Juni 2010.

[Bu97] Buttazzo, G. C.: *Hard Real-Time Computing Systems. Predictable Scheduling, Algorithms and Applications.* Boston; Dordrecht; London, Kluwer Academic Publishers, 1997.

[Co02] Cottet, F.; Delacroix, J.; Kaiser, C.; Mammeri, Z.: *Scheduling in Real-Time Systems.* Chichester, John Wiley & Sons Ltd., 2002.

[Co09] Coxon, A.: *FPGAs auf Low Power trimmen. Verfahren zum Optimieren von FPGA-Designs auf geringstmögliche Verlustleistung.* elektronik industrie, Hüthig Verlag, Ausgabe 1/2, 2009.

[Cr93] Cray Research, Inc.: *Cray T3D System Architecture Overview*. Handbuch HR-04033, 1993.

[Cu07] Curd, D.: *Power Consumption in 65 nm FPGAs.* Xilinx Inc., Whitepaper WP246, Februar 2007.

[De06] Derutin, J. P.; Damez, L.; Desportes, A.; Lazaro Galilea, J.L.: *Design of a Scalable Network of Communicating Soft Processors on FPGA*. CAMP 2006: International Workshop on Computer Architecture for Machine Perception and Sensing, Montréal; Québec, August 2006, S. 184-189.

[Do05] Donner, A.: *Ersatzteilversorgung: ungewiss!* erschienen in Elektronik & Entwicklung, Ausgabe November 2005, S. 16-18.

[Do10] Dorsey, P.: *Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency.* Xilinx White Paper WP380, Oktober 2010.

[Du03] Duato, J.; Yalamanchili, S.; Ni, L.: *Interconnection Networks. An Engineering Approach.* San Francisco, Morgan Kaufmann, 2003.

[Eb07] Eberhard, D.: *Kapselung sicherheitskritischer Funktionen in automobilen Steuergeräten.* erschienen in: Holleczek, P.; Vogel-Heuser, B. (Hrsg.): *Mobilität und Echtzeit.* Informatik aktuell, Heidelberg; Berlin, Springer Verlag, 2007, S. 107-116.

[Fi10] Fischer, R.: *Methodik für die Verlustleistungsabschätzung von Prozessoren mit Pipeline-Strukturen.* Dissertation, Universität der Bundeswehr München, Mai 2010.

[Fl99] Flynn, M. J., Hung, P., Rudd K. W.: *Deep-Submicron Microprocessor Design Issues.* IEEE Micro, Vol. 19, Nr. 4, Juli/August 1999, S. 11-22.

[Ga02] Garcia, A.; Pérez, L.; Acuña, R.: *Power consumption management on FPGAs.* Proceedings of the 15th International Conference on Electronics, Communications and Computers (CONIELECOMP 2005), Februar 2005.

[Ge07] Gessler, R.; Mahr, T.: *Hardware-Software-Codesign. Entwicklung flexibler Mikroprozessor-FPGA-Hochleistungssysteme.* Wiesbaden, Vieweg Verlag, 2007.

[Gr96] Gropp, W.; Lusk, E.; Skjellum, A.: *Using MPI.* Cambridge; Massachusetts, The MIT Press, 1996.

[Gr00] Grammatikakis, M. D.; Hsu, D. F.; Kraetzl, M.: *Parallel System Interconnections and Communications.* Boca Raton; London; New York; Washington, CRC Press Inc., 2000.

[Ha03] Hagen, M.: *Methoden, Daten- und Prozessmodell für das Ersatzteilmanagement in der Automobilelektronik.* Dissertation, Fakultät Maschinenwesen der Technischen Universität Dresden, Januar 2003.

[He78] Henn, R.: *Antwortzeitgesteuerte Prozessorzuteilung unter strengen Zeitbedingungen.* aus Computing, Vol. 19, Wien, Springer Verlag, September 1978, S. 209-220.

[Hu05] Huerta, P.; Castillo, J.; Mártinez, J. I.; López, V.: *A MicroBlaze based Multiprocessor SoC.* WSEAS Transactions on Circuits and Systems, Athen; Griechenland, Vol. 4, Nr. 5, Juli 2005, S. 423–430.

[Kb09] Kraftfahrt-Bundesamt: *Fahrzeugzulassungen. Bestand Fahrzeugalter.* Statistik des Kraftfahrt-Bundesamtes, Stand: 01.01.2009.

[Kl05] Klein, M.: *Static Power and the Importance of Realistic Junction Temperature Analysis.* Xilinx Inc., Whitepaper WP221, März 2005.

[Ki03] Kim, N.S.; Austin, T.; Baauw, D.; Mudge, T.; Flautner, K.; Hu, J.S.; Irwin, M.J.; Kandemir, M.; Narayanan, V.: *Leakage Current: Moore's Law Meets Static Power.* IEEE Computer, Vol. 36, Nr. 12, Dezember 2003, S. 68-75.

[Ki09] Kindel, O.; Friedrich, M.: *Softwareentwicklung mit AUTOSAR. Grundlagen, Engineering, Management in der Praxis.* 1. Auflage, Heidelberg, dpunkt.verlag, 2009.

[Ku02] Kumar, S.; Jantsch, A.; Soininen, J.-P.; Forsell, M.; Millberg, M.; Öberg, J.; Tiensyrjä, K.; Hemani, A.: *A Network on Chip Architecture and Design Methodology.* Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2002, S. 105–112.

[La02] Lackey, D. E.; Zuchowski, P. S.; Bednar, T. R.; Stout, D. W.; Gould, S. W.; Cohn, J. M.: *Managing power and performance for System-on-Chip designs using Voltage Islands.* Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design (ICCAD ´02), San Jose, November 2002, S. 195-202.

[Le02] Leen, G.; Heffernan, D.: *Expanding automotive electronic systems.* IEEE Computer magazine, Vol. 35, Nr. 1, Januar 2002, S. 88-93.

[Le07] Lee, H. G.; Chang N.; Ogras, U. Y.; Marculescu, R.: *On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches.* ACM Transactions on Design Automation of Electronic Systems, Vol. 12, Nr. 3, Artikel 23, August 2007.

[Me06] Mentor Graphics: *Confronting Complexity.* Automotive Industry White Paper, Mai 2006.

[Mo65] Moore, G. E.: *Cramming more components onto integrated circuits.* Electronics, Vol. 38, Nr. 8, 19. April 1965.

[Mp08] Mplemenos, G.-G.; Papaefstathiou, I.: *MPLEM: An 80-processor FPGA Based Multiprocessor System.* FCCM '08: 16th International Symposium Field-Programmable Custom Computing Machines, April 2008, S. 273–274.

[Na05] National Semiconductor: *LM1086 Data Sheet.* Dokumentation DS100948, 2005.

[Na81] Nassimi, D.; Sahni, S.: *A Self-Routing Benes Network and Parallel Permutation Algorithms.* IEEE Transactions on Computers, Vol. C-30, Nr. 5, Mai 1981, S. 332-340.

[Ni05] Niyogi, K.; Marculescu, D.: *Speed and Voltage Selection for GALS Systems Based on Voltage/Frequency Islands.* ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation, Shanghai, 2005, S. 292-297.

[No06] Nolte, T.: *Share-Driven Scheduling of Embedded Networks.* Dissertation, Universität Mälardalen, Mai 2006.

[OM08] Object Management Group: *Software Process Engineering Meta-Model.* Version 2.0, April 2008.

[Op71] Opferman, D. C.; Tsao-Wu, N. T.: On a Class of Rearrangeable Switching Networks. Part I: Control Algorithm. The Bell System Technical Journal, Vol. 50, Nr. 5, Mai-Juni 1971, S. 1579-1600.

[Pi96] Piguet, C.; Schneider, T.; Masgonty, J.-M.; Arm, C.; Durand, S.; Stegers, M.: *Low-Power Embedded Microprocessor Design.* Proceedings of the 22nd EUROMICRO Conference, Prag, September 1996, S. 600-605.

[Pl06] PLDApplications: *XpressFX Reference Manual.* 2006.

[Ra90] Raghavendra, C. S.; Boppana, R.: *Optimal Self-Routing of Linear-Complement Permutations in Hypercubes.* Proceedings of the Fifth Distributed Memory Computing, IEEE CS Press, 1990, S. 800-808.

[Ra91] Raghavendra, C.S.; Boppana, R.V.: *On Self-Routing in Benes and Shuffle-Exchange Networks.* IEEE Transactions on Computers, Vol. 40, Nr. 9, September 1991, S. 1057-1064.

[Ra07] Rauber, T.; Rünger, G.: *Parallele Programmierung.* 1. Auflage, Berlin; Heidelberg; New York, Springer Verlag, 2007.

[Ra10] Racu, R.: *The role of timing analysis in automotive network design.* 4. Symtavision NewsConference, Braunschweig, September 2010.

[Re09] Reif, K.: *Automobilelektronik. Eine Einführung für Ingenieure.* 3. Auflage, Wiesbaden, Vieweg+Teubner Verlag, 2009.

[Ri87] Richter, H.: *Multiprozessor mit dynamisch variabler Topologie.* Dissertation, Fakultät für Elektrotechnik und Informationstechnik, Technische Universität München, 1987.

[Ri92] Richter, H.: *Interconnection Network.* U.S. Patent Nr. 5.175.539, 1992.

[Ri97] Richter, H.: *Verbindungsnetzwerke für parallele und verteilte Systeme.* Heidelberg; Berlin; Oxford, Spektrum Akademischer Verlag, 1997.

[Ro03] Royal, A.; Cheung, P.: *Globally asynchronous locally synchronous FPGA architectures.* Proceedings of 13th Field Programmable Logic and Applications, Lecture Notes in Computer Science, Vol. 2778, Heidelberg; Berlin, Springer Verlag, 2003, S. 355-364.

[Rz94] Rzehak, H.: *Echtzeitsysteme und Fuzzy Control. Konzepte, Werkzeuge, Anwendungen.* Braunschweig; Wiesbaden, Vieweg Verlag, 1994.

[Sa93] Sautter, D.; Weinerth, H.: *Lexikon Elektronik und Mikroelektronik.* 2. aktualisierte und erweiterte Auflage, Springer Verlag, Berlin, 1993.

[Sc05a] Scholz, P.: *Softwareentwicklung eingebetteter Systeme. Grundlagen, Modellierung, Qualitätssicherung.* Berlin; Heidelberg; New York, Springer Verlag, 2005.

[Sc05b] Schildt, G. H.; Kahn, D.; Kruegel, C.; Moerz, C.: *Einführung in die Technische Informatik.* 2. Auflage, Wien; New York, Springer Verlag, 2005.

[Sh02] Shang, L.; Kaviani, A.S.; Bathala, K.: *Dynamic Power Consumption in Virtex™-II FPGA Family.* FPGA `02, Proceedings of the 2002 ACM/SIGDA tenth international symposium on field-programmable gate arrays, Monterey, Februar 2002, S. 157-164.

[Si05] Silberschatz, A.; Gagne, G.; Galvin, P. B.: *Operating System Concepts*. 7. Auflage, Hoboken, John Wiley & Sons Inc., 2005.

[Su04] Sutter, G.; Todorovich, E.; Boemo E.: *Design of Power Aware FPGA-based Systems.* Jornadas de Computación Reconfigurable y Aplicaciones, JCRA 2004, Barcelona, September 2004.

[Ta09] Tanenbaum, A. S.: *Moderne Betriebssysteme.* 3. Auflage, München, Pearson Studium, 2009.

[Te05] Telikepalli,A.: *Power vs. Performance: The 90 nm Inflection Point.* Xilinx Whitepaper WP223, Mai 2005.

[Th11] Thordsen, F.: *Fachartikel: Fahrzeugalter.* Statistik des Kraftfahrt-Bundesamtes zum Fahrzeugalter, Stand: 15.04.2011.

[Vw08] Volkswagen AG: *Der Golf 2009.* Selbststudienprogramm Nr. 423, September 2008.

[We09] Wetzel, H.-J.: *Das physische Bordnetz als Architekturtreiber.* Bordnetze, München, November 2009.

[Wi09] Wille, M.: *CarRing II: Entwurf und exemplarische Implementierung der Schichten 2 bis 6 des OSI-7-Schichtenmodells für ein zuverlässiges Echtzeit-Kommunikationsnetzwerk im Automobil.* Dissertation, Technische Universität Clausthal, 2009.

[Wö05] Wörn, H.; Brinkschulte, U.: *Echtzeitsysteme. Grundlagen, Funktionsweisen, Anwendungen.* Berlin; Heidelberg; New York, Springer Verlag, 2005.

[Wu80] Wu, C.-L.; Feng, T.-Y.: *On a Class of Multistage Interconnection Networks.* IEEE Transactions on Computers, Vol. C-29, Nr. 8, August 1980, S. 694-702.

[Wu11] Wu, X., Gopalan, P., Lara, G.: *Xilinx Next Generation 28 nm FPGA Technology Overview.* Xilinx Whitepaper WP312, Version 1.1, März 2011.

[Xi02] Xilinx: *Embedded System Tools Reference Manual.* Xilinx Dokumentation Embedded Development Kit EDK 10.1, 2002.

[Xi05] Xilinx: *Spartan-3 Starter Kit Board. User Guide.* Xilinx Dokumentation UG130, 2005.

[Xi08a] Xilinx: *ML505/ML506/ML507 Evaluation Platform. User Guide.* Xilinx Dokumentation UG347, 2008.

[Xi08b] Xilinx: *MicroBlaze Processor Reference Guide. Embedded Development Kit EDK 10.1i.* Xilinx Dokumentation UG081, Version 9.0, 2008.

[Xi10] Xilinx: *LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c).* Xilinx Product Specification DS449, April 2010.

[Xi11a] Xilinx: *Platform Specification Format Reference Manual. Embedded Development Kit (EDK) 13.1.* Xilinx Dokumentation UG642, März 2011.

[Xi11b] Xilinx: *7 Series FPGAs Overview.* Xilinx Advance Product Specification DS180, Version 1.7, Juli 2011.

[Xu08] Xu, S.; Pollitt-Smith, H.: *A Multi-MicroBlaze Based SOC System: From SystemC Modeling to FPGA Prototyping.* The 19th IEEE/IFIP International Symposium on Rapid System Prototyping, Monterey, Juni 2008, S. 121-127.

[Ya06] Yasuura, H.; Ishihara, T.; Muroyama, M.: *Energy Management Techniques for SoC Design.* Essential Issues in SoC Design, Springer Verlag, 2006, S. 177-223.

[Ye98] Yeap, G.: *Practical Low Power Digital VLSI Design.* Boston; Dordrecht; London, Kluwer Academic Publishers, 1998.

[Ze03] Zeferino, C. A.; Susin, A. A.: SoCIN: *A Parametric and Scalable Network-on-Chip.* Integrated Circuits and Systems Design (SBCCI) 2003, S. 169-174.

[Zi79] Ziegler, B.: *Prozessorzuteilung unter strengen Zeitbedingungen.* aus Computing, Vol. 23, Wien, Springer Verlag, März 1979, S. 31-41.

[Zi08] Zimmermann, W.; Schmidgall, R.: *Bussysteme in der Fahrzeugtechnik.* 3. Auflage, Wiesbaden, Vieweg+Teubner Verlag, 2008.

[Zv02] ZVEI: *Langzeitversorgung der Automobilindustrie mit elektronischen Baugruppen.* Weißbuch des Zentralverbandes Elektrotechnik- und Elektronikindustrie e.V. (ZVEI), Juni 2002.

Abbildungen

Tabellen

Anhang A - Implementierung im FPGA

A.1 Entwicklungsumgebung

Xilinx Platform Studio (XPS). Als Entwicklungsumgebung für ConPar wurde das *Embedded Development Kit (EDK)* der Fa. Xilinx in der Version 10.1 verwendet. Das darin enthaltene *Xilinx Platform Studio (XPS)* bietet alle Aktionen zur Erstellung eines MPSoC in einem einzigen Tool. Neben der baukastenartigen Konstruktion von Prozessorarchitekturen ist es im EDK möglich, eigene VHDL-Module (IP-Cores) und Softwareapplikationen (C, C++) zu erstellen, was die Zeit beim Entwickeln und Testen von Änderungen stark verkürzt. Der Nachteil dieser Methode besteht darin, dass das EDK nur auf einem 32-Bit Betriebssystem ausgeführt werden kann, wodurch leicht Speicherengpässe bei der Synthese aufgrund der limitierten Adressierung des Arbeitsspeichers (≤ 4 GiB) auftreten können.

Integrated Software Environment (ISE). Bereits in der EDK Version 10.1 gab es die Empfehlung, neue Projekte aus dem *Integrated Software Environment (ISE)* heraus zu erstellen. Die ISE dient in erster Linie zur Erstellung und Synthese von VHDL-Code, weshalb die Prozessorarchitektur im ISE-Projekt als *Embedded Processor* angelegt und durch den externen Aufruf im EDK bearbeitet wird. Hieraus ergibt sich der Nachteil, dass zwei verschiedene Werkzeuge für ein Projekt verwendet werden müssen. Auf der anderen Seite kann die ISE auch auf einem 64-Bit Betriebssystem ausgeführt werden, wodurch deutlich mehr Arbeitsspeicher adressiert werden kann.

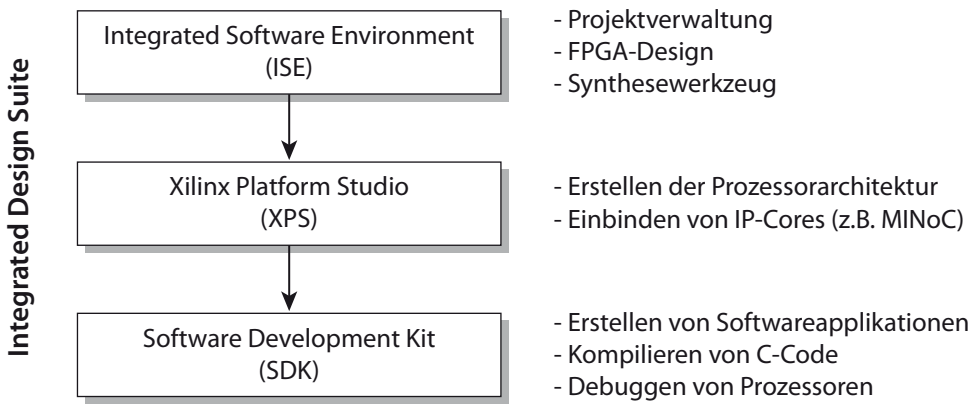


Abbildung A.1: Entwicklungswerkzeuge für Xilinx FPGAs

Software Development Kit (SDK). Mit der EDK Version 10.1 gab es ebenfalls die Empfehlung, die Softwareapplikation im *Software Development Kit (SDK)* zu erstellen. Während es im EDK Version 10.1 noch möglich war, Softwareapplikationen im XPS und somit außerhalb des SDK zu erstellen und zu kompilieren, wurde diese Möglichkeit in den nachfolgenden Versionen leider eliminiert.

A.2 Multiprozessorsystem

A.2.1 Prozessoren

Softprozessoren. Als Prozessoren wurden im MPSoC Softprozessoren vom Typ MicroBlaze eingesetzt. Der MicroBlaze ist ein proprietärer 32-Bit Softprozessor der Firma Xilinx in RISC-Architektur [Xi08b]. FPGA-basierte Softprozessoren wie der MicroBlaze bieten den großen Vorteil, dass sie als VHDL-Code vorliegen. Dadurch können auf einem einzigen Chip theoretisch beliebig viele Softprozessoren implementiert werden. Die maximale Anzahl an Softprozessoren auf einem einzelnen FPGA wird lediglich durch die verfügbaren Ressourcen auf dem Chip begrenzt. Darüber hinaus können Softprozessoren sehr leicht an die Erfordernisse

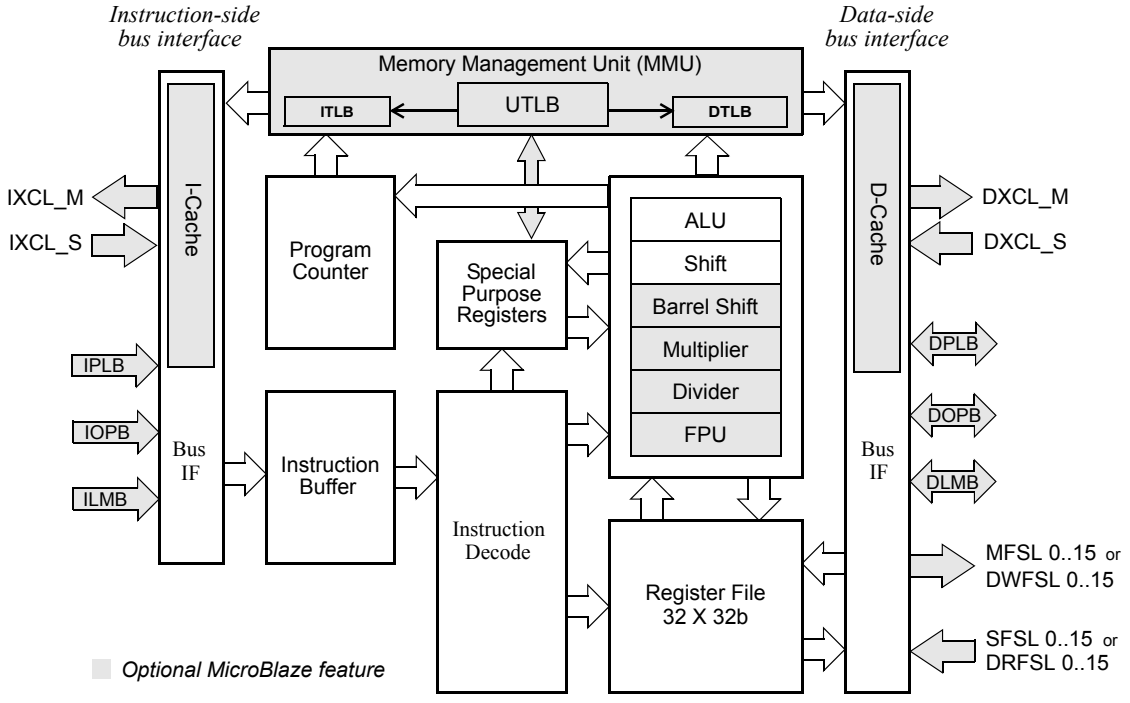


Abbildung A.2: MicroBlaze Core Block Diagram[1]

[1] grafische Darstellung aus [Xi08b, S. 10]

der Applikation angepasst werden[2]. So kann im XPS für jeden MicroBlaze im MPSoC mit wenigen Mausklicks eine geeignete Konfiguration (Interfaces, Recheneinheiten, etc.) gewählt werden.

Aufbau. Die **Abbildung A.2** zeigt die einzelnen Komponenten des Xilinx MicroBlaze im Blockdiagramm. Die weiß hinterlegten Elemente gehören zur Basisversion und sind damit unveränderlich. Allerdings kann z.B. die Größe der Befehlspipeline von 3 auf 5 Stufen erweitert werden, was in der Abbildung nicht explizit dargestellt ist. Die grau hinterlegten Elemente können im EDK per Mausklick für jeden Prozessor einzeln an- bzw. abgewählt werden, wodurch jeder einzelne Prozessor im MPSoC an die Anforderungen der jeweiligen Softwareapplikation angepasst werden kann. Bei der Konfiguration der Prozessoren muss ein Kompromiss zwischen dem Ressourcenbedarf (z.B. Chipfläche[3], Verlustleistung[4]) und der in der Applikation erforderlichen Rechenleistung gefunden werden.

Programmierung. Die Programmierung der MicroBlaze Softprozessoren erfolgt entweder im XPS[5] oder im SDK. In den beiden genannten Entwicklungsumgebungen steht jeweils ein C-Compiler sowie ein Debugging-Tool auf GNU-Basis zur Verfügung. Als Betriebssystem wurde in der prototypischen Implementierung von *ConPar* das *Stand-Alone* von Xilinx verwendet, eine sehr einfache und ressourcenschonende Plattform mit den wichtigsten Funktionen (z.B. Ein-/Ausgabe).

A.2.2 Speicher

Lokalspeicher on-Chip. Bei Space-Sharing wird jedem Prozessor im MPSoC ein eigener Lokalspeicher zugewiesen, wodurch ein automatischer Speicherschutz ohne zusätzliches Speichermanagement gewährleistet ist[6]. Als Speichermodul wird das interne Block-RAM verwendet, welches auf dem FPGA-Chip integriert ist. Die Anbindung externer Speichermodule ist aufgrund der benötigten IOs nicht vorgesehen.

---

[2] vgl. 4.5.3 Softprozessoren, S. 67

[3] vgl. Abbildung 4.13: Anteiliger Flächenbedarf im Softprozessor, S. 67

[4] vgl. Abbildung 6.13: Anteiliger Energiebedarf verschiedener Komponenten im Softprozessor am Beispiel des Xilinx MicroBlaze, S. 187

[5] nur bis zur Version 10.1 möglich (vgl. A.1 Entwicklungsumgebung, S. 249)

[6] vgl. 4.4.2 Räumliche Isolation durch Space-Sharing, S. 59

Harvard-Architektur. Die Anbindung des Speichers an den Prozessor erfolgt durch den Local Memory Bus (LMB). Beim verwendeten Xilinx MicroBlaze ist der Speicher als Harvard-Architektur ausgeführt[7], weshalb jeder Prozessor je eine getrennte Speicherbusanbindung für Befehle (ILMB) und Daten (DLMB) besitzt (vgl. Abbildung A.2).

A.2.3 Kommunikation

Fast Simplex Link (FSL). Für die Interprozessorkommunikation innerhalb des MPSoCs verfügt jeder MicroBlaze über ein eigenes FSL-Interface[8]. Über dieses Interface kann eine sehr schnelle unidirektionale Verbindung zu einem anderen Prozessor aufgebaut werden, um Nachrichten zu übertragen. Die Vermittlung der FSL-Verbindungen erfolgt über ein eigenes Verbindungsnetzwerk (MINoC), welches im Rahmen der vorliegenden Arbeit entwickelt und als IP-Core auf VHDL-Basis implementiert wurde[9].

A.2.4 Ein- /Ausgabe

IO-Module. Für die Kommunikation außerhalb des MPSoCs verfügt jeder MicroBlaze über einen eigenen Prozessorbus, an den zusätzliche IP-Cores angebunden werden können. Hierfür können fertige Module für digitale Ein- und Ausgänge (General Purpose IO) oder Busanbindungen wie SPI oder I^2C verwendet werden. Darüberhinaus besteht aber auch die Möglichkeit, eigene IP-Cores einzubinden.

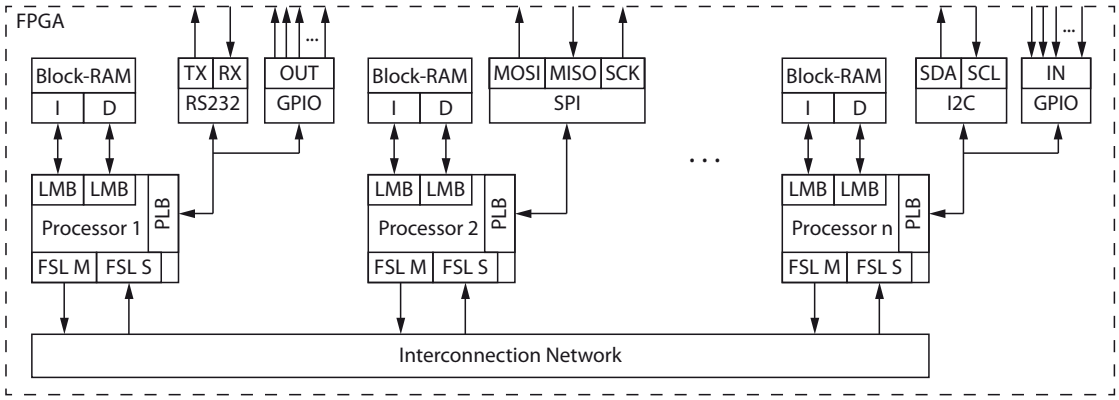


Abbildung A.3: Aufbau des Multiprozessorsystems mit Beispiel-IOs

[7] vgl. 4.5.4 Speicher, S. 69 f.

[8] Fast Simplex Link (FSL) [vgl. Xi10]

[9] vgl. 5.6 Echtzeit-Netz, S. 124 ff.

A.3 Verbindungsnetzwerk

A.3.1 Kommunikationskanal

Fast Simplex Link (FSL). Als Kommunikationskanal für die Interprozessorkommunikation wurde der proprietäre *Fast Simplex Link (FSL)* der Firma Xilinx genutzt. Der FSL bietet eine sehr schnelle unidirektionale Verbindung zwischen zwei Prozessoren und eignet sich daher ausgezeichnet als Kommunikationskanal in einem mehrstufigen Netz. Da eine FSL-Verbindung per Spezifikation jeweils nur einen Sender und einen Empfänger besitzt, ist keine Arbitrierung für den Verbindungsaufbau notwendig. Im Gegenzug müssen bei der Realisierung einer Broadcastkommunikation zusätzliche Mechanismen im Netzwerk implementiert werden, um mehrere Empfänger anzusprechen. Die Interprozessorkommunikation kann synchron oder durch den Einsatz von FIFO-Speichern auch asynchron erfolgen, wodurch ein GALS-Design[10] ermöglicht wird. Auf der Netzwerkseite müssen so viele FSL-Interfaces wie Softprozessoren implementiert werden, während auf der Prozessorseite ein FSL-Interface pro Prozessor genügt.

FSL-Interface. Die **Abbildung A.4** zeigt die Signale der FSL-Interfaces von Sender und Empfänger. Die mit *FSL_M* gekennzeichneten Signale auf der linken Seite gehören zum Sender der Daten, der als Master bezeichnet wird. Die mit *FSL_S* gekennzeichneten Signale auf der rechten Seite gehören zum Empfänger der Daten, der als Slave bezeichnet wird.

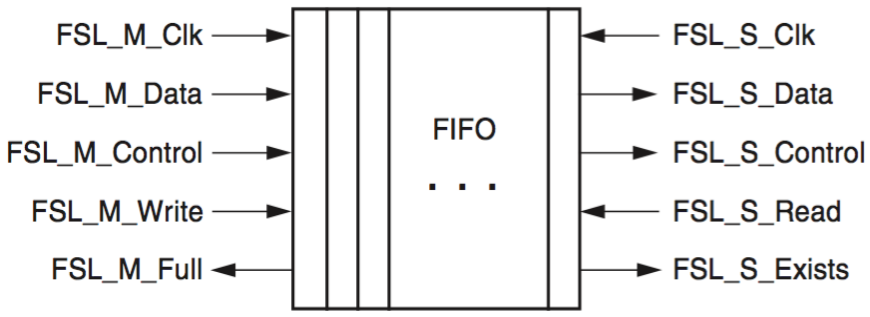


Abbildung A.4: Fast Simplex Link (FSL) - Blockschaltbild [Xi10, S. 2]

Datenübertragung. In der **Abbildung A.5** und der **Abbildung A.6** sind die Signalverläufe auf der Senderseite (Daten schreiben) und der Empfängerseite (Daten lesen) während der Datenübertragung dargestellt. Wichtig für eine Broadcastübertragung sind die Signale *FSL_M_Full* und *FSL_S_Read*, da diese der Handshake-

[10] vgl. 6.3.2 GALS-Design, S. 190

kommunikation dienen und hier bei mehreren Empfänger gesammelt werden müssen. Für das Verbindungsnetzwerk ist neben den Handshakesignalen das Signal *FSL_M_Control* von besonderer Bedeutung, da hierüber die Steuerdaten für den Aufbau der Verbindung gesendet werden.

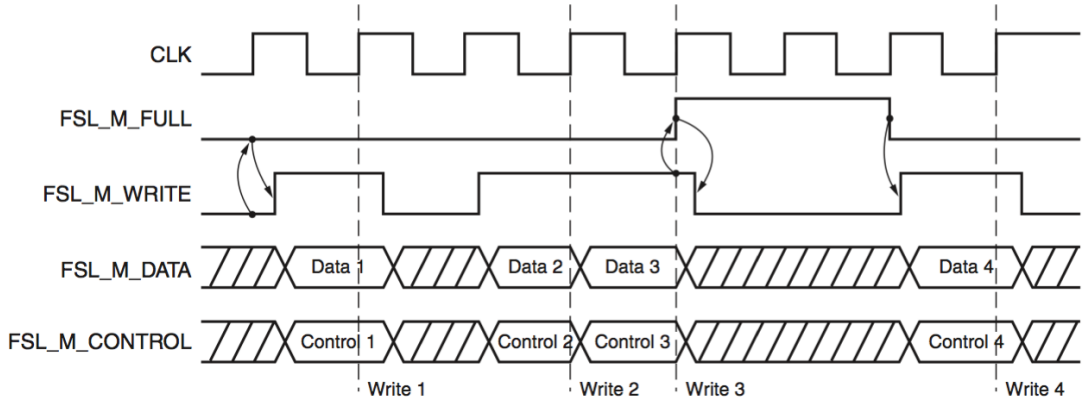


Abbildung A.5: Fast Simplex Link (FSL) - Daten schreiben [Xi10, S. 5]

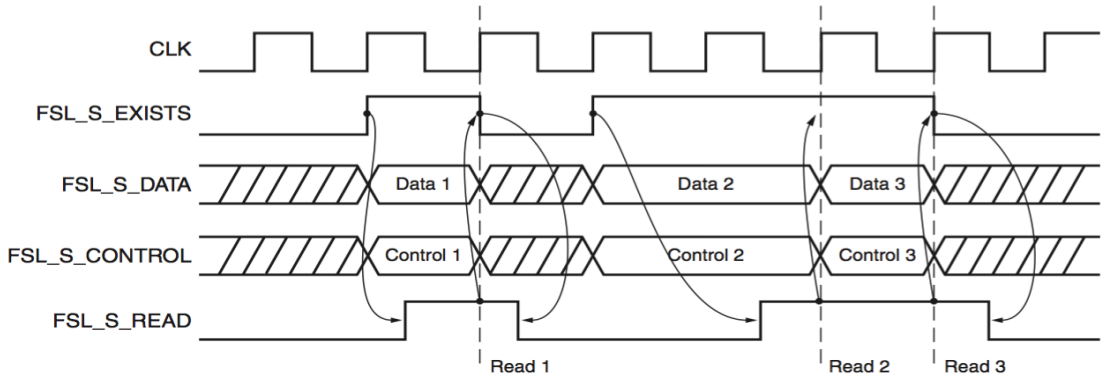


Abbildung A.6: Fast Simplex Link (FSL) - Daten lesen [Xi10, S. 5]

A.3.2 Mehrstufiges Netzwerk als IP-Core

IP-Core als Template erstellen. Das Verbindungsnetzwerk für die Interprozessorkommuniaktion ist als mehrstufiges Netzwerk ausgeführt und wird als benutzerdefinierter IP-Core implementiert. Hierzu wird im XPS zunächst ein neuer IP-Core als Template erstellt. Dabei empfiehlt es sich, den IP-Core im aktuellen Projektverzeichnis abzulegen. Als Namen habe ich *min* (für Multistage Interconnection Network) gewählt. Die wichtigste Einstellung ist die Auswahl des FSL-Interfaces, da FSL als Kommunikationskanal für das Verbindungsnetzwerk dient. Alle anderen Einstellungen sind dagegen ohne Bedeutung. Als Ergebnis hat man nun im IP-Katalog im Ordner *USER* einen neuen IP-Core als Template erstellt.

Anzahl der FSL-Schnittstellen erhöhen. Nachdem das Template erstellt ist, muss nun die Anzahl der FSL-Schnittstellen auf die Anzahl der Prozessoren im MPSoC erhöht werden. Dies geht leider nicht ohne zusätzlichen Aufwand, da Xilinx pro IP-Core nur ein FSL-Interface vorsieht. Aus diesem Grund müssen weitere Interfaces durch manuelles Editieren der Datei *.mpd*[11] hinzugefügt werden. Diese Datei befindet sich im Projektverzeichnis unter *\pcores* im Verzeichnis *\data* des IP-Cores und kann direkt im XPS geöffnet und editiert werden. Für jedes FSL-Interface muss ein eindeutiger Name vergeben werden, wobei die Semantik des Bezeichners keine Rolle spielt. In der prototypischen Implementierung von *ConPar* habe ich den Namen AUx gewählt. Das Listing A.1 zeigt die Deklaration der FSL-Interfaces AU0 und AU1 für den Anschluss der MicroBlazes 0 und 1 in der Datei *min.mpd*. Die Deklaration weiterer FSL-Interfaces erfolgt analog.

Listing A.1: FSL-Interfaces AU0 und AU1 in der Datei min.mpd

```
BEGIN min
## Bus Interfaces
BUS_INTERFACE BUS = AU0_SFSL, BUS_TYPE = SLAVE, BUS_STD = FSL
BUS_INTERFACE BUS = AU0_MFSL, BUS_TYPE = MASTER, BUS_STD = FSL
BUS_INTERFACE BUS = AU1_SFSL, BUS_TYPE = SLAVE, BUS_STD = FSL
BUS_INTERFACE BUS = AU1_MFSL, BUS_TYPE = MASTER, BUS_STD = FSL
...
## Ports
PORT FSL_Clk = „", DIR = I, SIGIS = Clk, BUS =
AU0_SFSL:AU0_MFSL:AU1_SFSL:AU1_MFSL
PORT FSL_Rst = OPB_Rst, DIR = I , SIGIS = RST,
BUS = AU0_SFSL:AU0_MFSL:AU1_SFSL:AU1_MFSL
PORT AU0_FSL_S_Clk = FSL_S_Clk, DIR = O, SIGIS = Clk, BUS = AU0_SFSL
PORT AU0_FSL_S_Read = FSL_S_Read, DIR = O, BUS = AU0_SFSL
PORT AU0_FSL_S_Data = FSL_S_Data, DIR = I, VEC = [0:31], BUS = AU0_SFSL
PORT AU0_FSL_S_Control = FSL_S_Control, DIR = I, BUS = AU0_SFSL
PORT AU0_FSL_S_Exists = FSL_S_Exists, DIR = I, BUS = AU0_SFSL
PORT AU0_FSL_M_Clk = FSL_M_Clk, DIR = O, SIGIS = Clk, BUS = AU0_MFSL
PORT AU0_FSL_M_Write = FSL_M_Write, DIR = O, BUS = AU0_MFSL
PORT AU0_FSL_M_Data = FSL_M_Data, DIR = O, VEC = [0:31], BUS = AU0_MFSL
PORT AU0_FSL_M_Control = FSL_M_Control, DIR = O, BUS = AU0_MFSL
PORT AU0_FSL_M_Full = FSL_M_Full, DIR = I, BUS = AU0_MFSL

PORT AU1_FSL_S_Clk = FSL_S_Clk, DIR = O, SIGIS = Clk, BUS = AU1_SFSL
PORT AU1_FSL_S_Read = FSL_S_Read, DIR = O, BUS = AU1_SFSL
PORT AU1_FSL_S_Data = FSL_S_Data, DIR = I, VEC = [0:31], BUS = AU1_SFSL
```

[11] Microprocessor Peripheral Definition, vgl. [Xi11a, S. 33 ff.]

```
PORT AU1_FSL_S_Control = FSL_S_Control, DIR = I, BUS = AU1_SFSL
PORT AU1_FSL_S_Exists = FSL_S_Exists, DIR = I, BUS = AU1_SFSL
PORT AU1_FSL_M_Clk = FSL_M_Clk, DIR = O, SIGIS = Clk, BUS = AU1_MFSL
PORT AU1_FSL_M_Write = FSL_M_Write, DIR = O, BUS = AU1_MFSL
PORT AU1_FSL_M_Data = FSL_M_Data, DIR = O, VEC = [0:31], BUS = AU1_MFSL
PORT AU1_FSL_M_Control = FSL_M_Control, DIR = O, BUS = AU1_MFSL
PORT AU1_FSL_M_Full = FSL_M_Full, DIR = I, BUS = AU1_MFSL
...
```

VHDL-Code implementieren. Die eigentliche Funktionalität des Netzwerks wird im IP-Core als VHDL-Code implementiert. Hierfür muss im XPS die Datei *.vhd* geöffnet werden, welche sich im Projektverzeichnis unter *\pcores* im Verzeichnis *\hdl* des IP-Cores befindet. Hier befindet sich bereits automatisch generierter Beispielcode aus dem Template, welcher durch den VHDL-Code des Verbindungsnetzwerks ersetzt werden muss.

Einbinden in das MPSoC. Nun steht das mehrstufige Verbindungsnetzwerk als IP-Core im IP-Katalog von XPS zur Verfügung und kann in das System eingefügt werden. Anschließend müssen in der Registerkarte *Bus Interfaces* die FSL-Anschlüsse des Netzwerks mit den MicroBlazes und in der Registerkarte *Ports* die Signale für *Clock* und *Reset* verdrahtet werden. Weitere Einstellungen sind am Netzwerk nicht erforderlich, da Steuerung, Takt und Datenaustausch ausschließlich über die FSL-Verbindungen erfolgen.

A.4 Interprozessorkommunikation

A.4.1 Asynchrone Kommunikation

Zeitliche Isolation. Durch Space-Sharing werden die Prozessoren im MPSoC zeitlich voneinander isoliert[12]. Dadurch ergibt sich die Möglichkeit, den Energiebedarf jedes einzelnen Prozessors anhand der erforderlichen Rechenleistung und dem Zustand des Programms zu optimieren[13].

Zeitliche Entkopplung der Kommunikation. Die notwendige Interprozessorkommunikation verbindet alle Prozessoren miteinander. Um die zeitliche Isolation der Prozessoren zu erhalten, muss die Kommunikation asynchron erfolgen. Diese

[12] 4.3.2 Zeitliche Isolation durch Space-Sharing, S. 56 f.

[13] vgl. 6.3.3 Dynamische Anpassung der Prozessorleistung, S. 190 ff.

Möglichkeit bietet der FSL. Durch den Einsatz von FIFO-Puffern im FSL-Kanal, sowohl auf der Sender- als auch auf der Empfängerseite, erfolgt eine zeitliche Entkopplung zwischen Sender und Empfänger der Nachricht. Die Größe der FIFO-Puffer kann im XPS nach Bedarf konfiguriert werden. Bei der prototypischen Implementierung von *ConPar* beträgt die Puffergröße 512 Byte.

Konfiguration. Die Konfiguration der FSL-Verbindungen erfolgt als Parameter im VHDL-Modul der jeweiligen Instanz. Die VHDL-Module der FSL-Verbindungen befinden sich im Projektverzeichnis unter *\hdl*. Das Listing A.2 zeigt die wichtigsten Konfigurationsdaten der FSL-Verbindungen für die Interprozessorkommunikation im MPSoC.

Listing A.2: Konfiguration der FSL-Verbindung

```
-- fsl_v20_00_wrapper.vhd
...
begin
   fsl_v20_00 : fsl_v20
      generic map (
         C_ASYNC_CLKS => 1,    -- asynchrone Kommunikation
         C_USE_CONTROL => 1,    -- Steuerkanal aktiv
         C_FSL_DEPTH => 512,    -- FIFO = 512 Byte
         ...
         );
...
```

A.4.2 Verbindungsaufbau

Übergabe der Zieladresse. Vor der Übertragung der Daten muss der Sender die Adresse des Empfängers an das Netzwerk übergeben. Die Zieladresse entspricht der ID des Zielprozessors, die bei Xilinx über den Wert der Variable *XPAR_MICROBLAZE_ID* abgefragt werden kann. Die Übergabe der Zieladresse vom Prozessor an das Netzwerk erfolgt über den Steuerkanal des FSL-Interfaces. Bei der Verwendung des Steuerkanals wird das Signal *FSL_M_Control* gesetzt, welches im Netzwerk zur Unterscheidung zwischen Steuer- und Sendedaten abgefragt wird.

Listing A.3: Übergabe der Zieladresse an das Netzwerk in C

```
// Zieladresse an Netzwerk senden
microblaze_bwrite_cntlfsl (target_id, channel);
```

Zur Auswertung der Zieladresse wird in dem von *FSL_Clk* aufgerufenen Prozess das Signal *FSL_M_Control* abgefragt.

Listing A.4: Auswertung der Steuerdaten im Netzwerk in VHDL

```
-- Auswertung der Zieladresse
if FSL_S_Exists = '1' and FSL_S_Control = '1' then
   -- Nummer des Zielprozessors auslesen
   Zieladresse := conv_integer(FSL_S_Data);
   -- weitere Logik zum Verbindungsaufbau
end if;
```

Falls die Zieladresse nicht bereits durch eine andere bestehende Verbindung belegt ist, kann die Übertragung der Daten zum Zielprozessor bereits im nächsten Takt von *FSL_Clk* erfolgen. Im anderen Fall wird die Übertragung verzögert, bis die Zieladresse wieder freigegeben wurde.

Routing der Verbindungen. Das Routing der Verbindungen im Netzwerk erfolgt durch kombinatorische Logik im Schaltnetz und außerhalb von taktgesteuerten Prozessen. Durch Boole`sche Verknüpfungen von Signalen auf VHDL-Basis[14] ist es möglich, das Routing aller Verbindungen im Netz unabhängig von der Netzgröße innerhalb einer Gatterlaufzeit durchzuführen. Diese Zeit liegt weit unterhalb einer Taktperiode, so dass bereits im nächsten Takt alle Schalterstellungen im dynamischen Netz gesetzt sind.

A.4.3 Datenübertragung

Datenübertragung. Da das mehrstufige Netzwerk eine direkte Verbindung vom Sender zum Empfänger der Daten herstellt, erfolgt die Übertragung der Daten analog einer 1:1-Verbindung. Die Daten des Sender werden durch das Netzwerk vom FIFO-Puffer des Senders in den FIFO-Puffer des Empfängers übertragen.

Listing A.5: Daten vom Sender an den Empfänger senden

```
// Daten senden
microblaze_bwrite_datafsl (*data, channel);
```

A.4.4 Verbindungsabbau

[14] vgl. 5.5.5.2 Problemstellung, S. 124

Senden der eigenen Adresse. Für den Abbau der Verbindung wird vom Prozessor die eigene Adresse als Zieladresse an das Netzwerk übergeben. Sind die Adressen von Sender und Empfänger identisch, so wird vom Netzwerk lediglich die bestehende Verbindung abgebaut. Im anderen Fall würde automatisch eine Verbindung zur neuen Zieladresse aufgebaut werden.

Listing A.6: Übergabe der eigenen Adresse an das Netzwerk in C

```
// Eigene Adresse an Netzwerk senden
microblaze_bwrite_cntlfsl (XPAR_MICROBLAZE_ID, channel);
```

A.4.5 Nachrichtenübertragung

Blocktransfer. Aus Gründen der Effizienz ist die Nachrichtenübertragung i. d. R. so gestaltet, dass eine Nachricht nicht aus einem einzelnem Datum besteht, sondern mehrere Daten als Block übertragen werden. Die **Abbildung A.7** zeigt als Beispiel den Aufbau einer CAN-Nachricht, wie sie für die Kommunikation zwischen den Steuergeräten im Fahrzeug verwendet wird. Da CAN seit vielen Jahren zur Kommunikation im Fahrzeug eingesetzt wird, soll das Senden und Empfangen von existierenden CAN-Nachrichten im MPSoC emuliert werden. In einer CAN-Nachricht können pro Nachricht bis zu 8 Byte an Nutzdaten übertragen werden. Zusätzlich enthält jede CAN-Nachricht ein Byte zur eindeutigen Identifikation der Nachricht sowie ein weiteres Byte zur Spezifikation der Nachrichtenlänge. Insgesamt werden demnach bei CAN pro Nachricht bis zu 10 Byte an Daten als Block übertragen. Die im CAN-Transportrahmen angehängte Prüfsumme kann bei einem NoC als obsolet betrachtet werden, da chip-interne Übertragungsfehler praktisch ausgeschlossen sind.

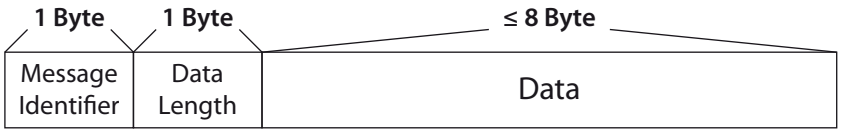


Abbildung A.7: Aufbau einer CAN-Nachricht

Integration der Datenstruktur von CAN. Der Inhalt des in Abbildung A.7 dargestellten Aufbaus einer CAN-Nachricht kann im Programm durch eine einfache Datenstruktur nachgebildet werden. Das **Listing A.7** zeigt den Aufbau einer Datenstruktur, welche die Elemente *ID*, *Länge* und *Daten* enthält. Damit lassen sich bei der Rezentralisierung von Steuergeräten bestehende CAN-Nachrichten einfach in

die Interprozessorkommunikation integrieren, ohne den Aufbau der Nachrichten verändern zu müssen.

Listing A.7: Datenstruktur einer CAN-Nachricht

```
struct message{
   int id;    // message identifier
   int length;    // data length
   int data[8];    // data bytes
};
```

Nachricht senden. Im Unterschied zu einem Bussystem, muss die Nachricht in einem Verbingungsnetzwerk an den Empfänger vermittelt werden. Das **Listing A.8** zeigt eine kleine Routine zum Senden eines Datenpakets. Um eine Nachricht im MPSoC zu senden, wird zunächst die Adresse des Empfängers an das Netzwerk übermittelt, damit die Verbindung aufgebaut werden kann. Anschließend werden die einzelnen Daten aus der Datenstruktur der Nachricht in einer Schleife seriell übermittelt. Nach dem Senden der Nachricht wird die Verbindung durch das Senden der eigenen Adresse abgebaut. Da die Kommunikation asynchron erfolgt, werden die Daten zunächst in den Sendepuffer des FSL geschrieben. Nach erfolgreichem Verbindungsaufbau überträgt das Verbindungsnetzwerk selbstständig die Daten vom Sendepuffer in den Empfangspuffer des Zielprozessors.

Listing A.8: Nachricht senden

```
void data_send (int target_id, int *data, int length){
   int a;
   /* Zieladresse über Steuerkanal an Netzwerk senden
   und Verbindung aufbauen */
   microblaze_bwrite_cntlfsl (target_id,0);
   // Nachricht über Datenkanal senden
   for (a = 0; a < length; a++){
      microblaze_bwrite_datafsl (*(data+a),0);
   };
   // Verbindung über Steuerkanal abbauen
   microblaze_bwrite_cntlfsl (XPAR_MICROBLAZE_ID,0);
   return;
}
```

Nachricht empfangen. Neue Nachrichten werden zunächst vom Verbindungsnetzwerk in den Empfangspuffer des FSL gelegt und dort später vom Zielprozessor ausgelesen. Das **Listing A.9** zeigt eine kleine Routine, um empfangene Nachrichten aus dem Empfangspuffer des FSL zu lesen. Hierbei wird zunächst der

FIFO-Puffer des FSL-Kanals auf den Erhalt neuer Daten geprüft. Sind keine neuen Daten vorhanden, so wird die Routine an dieser Stelle beendet und der Wert „0“ zurückgegeben. Im anderen Fall kann die Nachricht in einer seriellen Schleife in die Datenstruktur gelesen werden. In diesem Fall wird der Wert „1“ zurückgegeben.

Listing A.9: Nachricht empfangen

```
int data_receive (int *data, int length){
   int a = 0;
   // Empfangspuffer auf neue Nachricht testen
   microblaze_nbread_datafsl (*data,0);
   if (*data == 0) return(0);
   // neue Nachricht aus dem Empfangspuffer lesen
   for (a = 1; a < length; a++){
      microblaze_bread_datafsl (*(data+a),0);
   };
   return(1);
}
```

A.4.6 Routing

Routing-Algorithmus. Neben der Topologie beruht die Funktion des Netzwerks ganz wesentlich auf dem Routingverfahren[15]. Dazu wurde im FPGA ein Routing-Algorithmus als Schaltnetz implementiert, dessen Code nachfolgend in einem Beispiel beschrieben wird. Zum besseren Verständnis sind die verwendeten Bezeichnungen in der **Abbildung A.8** noch einmal im Netz dargestellt.

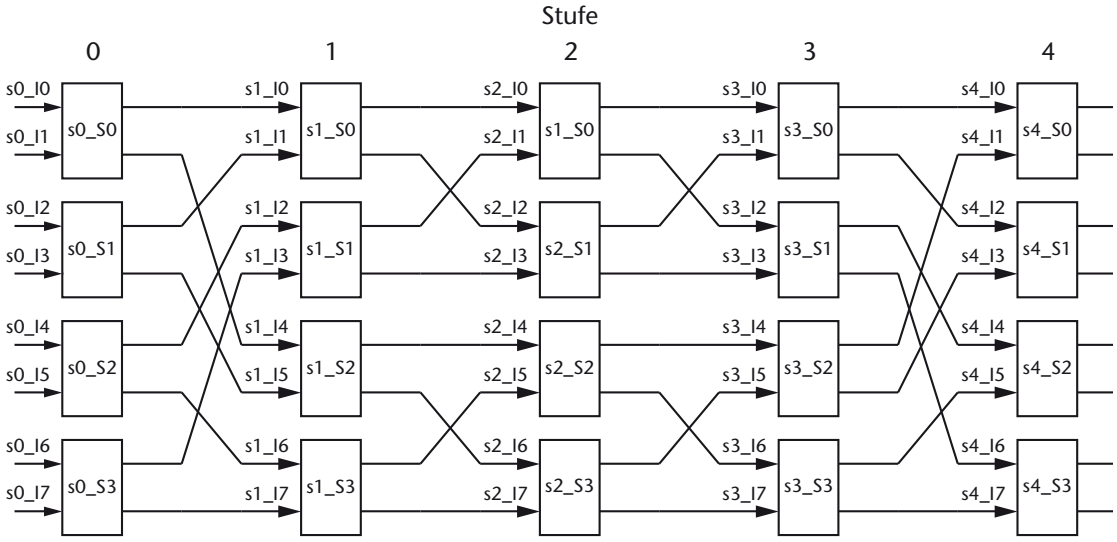


Abbildung A.8: Bezeichnungen für das Routing im Beneš-Netz

[15] Funktionsweise des Routingverfahrens in 5.5.5 Routing im Schaltnetz, S. 122 ff.

Zunächst werden die benötigten Signale deklariert, wobei die Schalterstellungen als Bits und die Zieladressen als Bitvektoren implementiert werden. Die Anordnung der Bits im Bit-Vektor ist in dem Beispiel wie folgt festgelegt:

- bit_vector(0 to 3) = $I(i_3 i_2 i_1 i_0)$

Listing A.10: Deklaration der benötigten Signale

```
architecture NETWORK of min is
...
-- Deklaration: Status Kreuzschalter
-- Bezeichner: s: Stufe, S: Schalter
signal s0_S0, s0_S1, s0_S2, s0_S3 : bit; -- Stufe 0
signal s1_S0, s1_S1, s1_S2, s1_S3 : bit; -- Stufe 1
signal s2_S0, s2_S1, s2_S2, s2_S3 : bit; -- Stufe 2
signal s3_S0, s3_S1, s3_S2, s3_S3 : bit; -- Stufe 3
signal s4_S0, s4_S1, s4_S2, s4_S3 : bit; -- Stufe 4
-- Deklaration: Signale für Routing-Algorithmus
-- Bezeichner: s: Stufe, I: Zieladresse
-- Schaltnetz
 -- Stufe 0
  -- Deklaration von der Zieladressen als Bitvektor [i3,i2,i1,i0]
  -- Schaltereingänge I
  signal s0_I0, s0_I1, s0_I2, s0_I3 : bit_vector(0 to 3);
  signal s0_I4, s0_I5, s0_I6, s0_I7 : bit_vector(0 to 3);
  -- Schalterausgänge I'
  signal s0_I0_1, s0_I1_1, s0_I2_1, s0_I3_1 : bit_vector(0 to 3);
  signal s0_I4_1, s0_I5_1, s0_I6_1, s0_I7_1 : bit_vector(0 to 3);
  -- Schalterausgänge I''
  signal s0_I0_2, s0_I1_2, s0_I2_2, s0_I3_2 : bit_vector(0 to 3);
  signal s0_I4_2, s0_I5_2, s0_I6_2, s0_I7_2 : bit_vector(0 to 3);
  -- Deklaration von g, k, f, S', k' und S als Bitsignal
  signal s0_g2, s0_g3, s0_g4, s0_g5 : bit;
  signal s0_k2, s0_k3, s0_k4, s0_k5, s0_k6, s0_k7 : bit;
  signal s0_f1, s0_f2, s0_f3 : bit;
  signal s0_S1_1, s0_S2_1, s0_S3_1 : bit;
  signal s0_k2_1, s0_k3_1, s0_k4_1, s0_k5_1 : bit;
 -- Stufe 1
  -- Deklaration von der Zieladressen als Bitvektor [i3,i2,i1,i0]
  -- Schaltereingänge I
  signal s1_I0, s1_I1, s1_I2, s1_I3 : bit_vector(0 to 3);
  signal s1_I4, s1_I5, s1_I6, s1_I7 : bit_vector(0 to 3);
  -- Schalterausgänge I'
  signal s1_I0_1, s1_I1_1, s1_I2_1, s1_I3_1 : bit_vector(0 to 3);
```

```
  signal s1_I4_1, s1_I5_1, s1_I6_1, s1_I7_1 : bit_vector(0 to 3);
 -- Stufe 2
  -- Deklaration von der Zieladressen als Bitvektor [i3,i2,i1,i0]
  -- Schaltereingänge I
  signal s2_I0, s2_I1, s2_I2, s2_I3 : bit_vector(0 to 3);
  signal s2_I4, s2_I5, s2_I6, s2_I7 : bit_vector(0 to 3);
  -- Schalterausgänge I'
  signal s2_I0_1, s2_I1_1, s2_I2_1, s2_I3_1 : bit_vector(0 to 3);
  signal s2_I4_1, s2_I5_1, s2_I6_1, s2_I7_1 : bit_vector(0 to 3);
 -- Stufe 3
  -- Deklaration von der Zieladressen als Bitvektor [i3,i2,i1,i0]
  -- Schaltereingänge I
  signal s3_I0, s3_I1, s3_I2, s3_I3 : bit_vector(0 to 3);
  signal s3_I4, s3_I5, s3_I6, s3_I7 : bit_vector(0 to 3);
  -- Schalterausgänge I'
  signal s3_I0_1, s3_I1_1, s3_I2_1, s3_I3_1 : bit_vector(0 to 3);
  signal s3_I4_1, s3_I5_1, s3_I6_1, s3_I7_1 : bit_vector(0 to 3);
 -- Stufe 4
  -- Deklaration von der Zieladressen als Bitvektor [i3,i2,i1,i0]
  -- Schaltereingänge I
  signal s4_I0, s4_I1, s4_I2, s4_I3 : bit_vector(0 to 3);
  signal s4_I4, s4_I5, s4_I6, s4_I7 : bit_vector(0 to 3);
  -- Schalterausgänge I'
  signal s4_I0_1, s4_I1_1, s4_I2_1, s4_I3_1 : bit_vector(0 to 3);
  signal s4_I4_1, s4_I5_1, s4_I6_1, s4_I7_1 : bit_vector(0 to 3);
```

Nachdem alle Signale deklariert sind, erfolgt nun das Routing der Schalterstufe 0.

Listing A.11: Routing der Schalterstufe 0

```
begin
-- Beginn Routing-Algorithmus
 -- Stufe 0
  -- 1. Vergleich
   -- Bestimmung von g
   s0_g2 <= '1' when ((s0_I2(1 to 2) = s0_I1_1(1 to 2)) and
               (s0_I2(0) = '1') and (s0_I1_1(0) = '1')) else '0';
   s0_g3 <= '1' when ((s0_I3(1 to 2) = s0_I0_1(1 to 2)) and
               (s0_I3(0) = '1') and (s0_I0_1(0) = '1')) else '0';
   s0_g4 <= '1' when ((s0_I4(1 to 2) = s0_I1_1(1 to 2)) and
               (s0_I4(0) = '1') and (s0_I1_1(0) = '1')) or
                     ((s0_I4(1 to 2) = s0_I3_1(1 to 2)) and
               (s0_I4(0) = '1') and (s0_I3_1(0) = '1') and
               (s0_f1 = '1')) else '0';
   s0_g5 <= '1' when ((s0_I5(1 to 2) = s0_I0_1(1 to 2)) and
               (s0_I5(0) = '1') and (s0_I2_1(0) = '1')) or
                     ((s0_I5(1 to 2) = s0_I2_1(1 to 2)) and
               (s0_I5(0) = '1') and (s0_I2_1(0) = '1') and
               (s0_f1 = '1')) else '0';
```

```
  -- Bestimmung von k
  s0_k2 <= '1' when ((s0_I2(1 to 2) = s0_I0_1(1 to 2)) and
              (s0_I2(0) = '1') and (s0_I0_1(0) = '1')) else '0';
  s0_k3 <= '1' when ((s0_I3(1 to 2) = s0_I1_1(1 to 2)) and
              (s0_I3(0) = '1') and (s0_I1_1(0) = '1')) else '0';
  s0_k4 <= '1' when ((s0_I4(1 to 2) = s0_I0_1(1 to 2)) and
              (s0_I4(0) = '1') and (s0_I0_1(0) = '1')) or
                    ((s0_I4(1 to 2) = s0_I2_1(1 to 2)) and
              (s0_I4(0) = '1') and (s0_I2_1(0) = '1') and
              (s0_f1 = '1')) else '0';
  s0_k5 <= '1' when ((s0_I5(1 to 2) = s0_I1_1(1 to 2)) and
              (s0_I5(0) = '1') and (s0_I1_1(0) = '1')) or
                    ((s0_I5(1 to 2) = s0_I3_1(1 to 2)) and
              (s0_I5(0) = '1') and (s0_I3_1(0) = '1') and
              (s0_f1 = '1')) else '0';
  s0_k6 <= '1' when ((s0_I6(1 to 2) = s0_I0_1(1 to 2)) and
              (s0_I6(0) = '1') and (s0_I0_1(0) = '1')) or
                    ((s0_I6(1 to 2) = s0_I2_1(1 to 2)) and
              (s0_I6(0) = '1') and (s0_I2_1(0) = '1') and
              (s0_f1 = '1')) or
                    ((s0_I6(1 to 2) = s0_I4_1(1 to 2)) and
              (s0_I6(0) = '1') and (s0_I4_1(0) = '1') and
              (s0_f2 = '1')) else '0';
  s0_k7 <= '1' when ((s0_I7(1 to 2) = s0_I1_1(1 to 2)) and
              (s0_I7(0) = '1') and (s0_I1_1(0) = '1')) or
                    ((s0_I7(1 to 2) = s0_I3_1(1 to 2)) and
              (s0_I7(0) = '1') and (s0_I3_1(0) = '1') and
              (s0_f1 = '1')) or
                    ((s0_I7(1 to 2) = s0_I5_1(1 to 2)) and
              (s0_I7(0) = '1') and (s0_I5_1(0) = '1') and
              (s0_f2 = '1')) else '0';
  -- Bestimmung von S'
  s0_S1_1 <= '1' when (s0_k2 = '1') or (s0_k3 = '1') else '0';
  s0_S2_1 <= '1' when (s0_k4 = '1') or (s0_k5 = '1') else '0';
  s0_S3_1 <= '1' when (s0_k6 = '1') or (s0_k7 = '1') else '0';
  -- Schalter markieren
  s0_f1 <= '1' when (s0_g2 = '1') or (s0_g3 = '1') or
                    (s0_k2 = '1') or (s0_k3 = '1') else '0';
  s0_f2 <= '1' when (s0_g4 = '1') or (s0_g5 = '1') or
                    (s0_k4 = '1') or (s0_k5 = '1') else '0';
  -- Schalterausgänge setzen
  s0_I0_1 <= s0_I0;
  s0_I1_1 <= s0_I1;
  s0_I2_1 <= s0_I2 when s0_S1_1 = '0' else s0_I3;
  s0_I3_1 <= s0_I2 when s0_S1_1 = '1' else s0_I3;
  s0_I4_1 <= s0_I4 when s0_S2_1 = '0' else s0_I5;
  s0_I5_1 <= s0_I4 when s0_S2_1 = '1' else s0_I5;
  s0_I6_1 <= s0_I6 when s0_S3_1 = '0' else s0_I7;
  s0_I7_1 <= s0_I6 when s0_S3_1 = '1' else s0_I7;
-- 2. Vergleich
  -- Bestimmung von k
  s0_k5_1 <= '1' when ((s0_I5_1(1 to 2) = s0_I7_2(1 to 2)) and
               (s0_I5_1(0) = '1') and (s0_I7_2(0) = '1')) else '0';
```

```
  s0_k4_1 <= '1' when ((s0_I4_1(1 to 2) = s0_I6_2(1 to 2)) and
                 (s0_I4_1(0) = '1') and (s0_I6_2(0) = '1')) else '0';
  s0_k3_1 <= '1' when ((s0_I3_1(1 to 2) = s0_I7_2(1 to 2)) and
                 (s0_I3_1(0) = '1') and (s0_I7_2(0) = '1')) or
                       ((s0_I3_1(1 to 2) = s0_I5_2(1 to 2)) and
                 (s0_I3_1(0) = '1') and (s0_I5_2(0) = '1')) else '0';
  s0_k2_1 <= '1' when ((s0_I2_1(1 to 2) = s0_I6_2(1 to 2)) and
                 (s0_I2_1(0) = '1') and (s0_I6_2(0) = '1')) or
                       ((s0_I2_1(1 to 2) = s0_I4_2(1 to 2)) and
                 (s0_I2_1(0) = '1') and (s0_I4_2(0) = '1')) else '0';
  -- Bestimmung von S
  s0_S0 <= '0';
  s0_S1 <= '1' when (s0_S1_1 = '1') or (s0_k2_1 = '1') or
                    (s0_k3_1 = '1') else '0';
  s0_S2 <= '1' when (s0_S2_1 = '1') or (s0_k4_1 = '1') or
                    (s0_k5_1 = '1') else '0';
  s0_S3 <= '1' when (s0_S3_1 = '1') else '0';
  -- Schalterausgänge setzen
  s0_I0_2 <= s0_I0;
  s0_I1_2 <= s0_I1;
  s0_I2_2 <= s0_I2 when s0_S1 = '0' else s0_I3;
  s0_I3_2 <= s0_I2 when s0_S1 = '1' else s0_I3;
  s0_I4_2 <= s0_I4 when s0_S2 = '0' else s0_I5;
  s0_I5_2 <= s0_I4 when s0_S2 = '1' else s0_I5;
  s0_I6_2 <= s0_I6 when s0_S3 = '0' else s0_I7;
  s0_I7_2 <= s0_I6 when s0_S3 = '1' else s0_I7;
 -- Verdrahtung Stufe 0 => Stufe 1 (Unshuffle Permutation)
 s1_I0 <= s0_I0_2;
 s1_I1 <= s0_I2_2;
 s1_I2 <= s0_I4_2;
 s1_I3 <= s0_I6_2;
 s1_I4 <= s0_I1_2;
 s1_I5 <= s0_I3_2;
 s1_I6 <= s0_I5_2;
 s1_I7 <= s0_I7_2;
```

Das Routing der Schalterstufe 0 ist damit abgeschlossen. Im nächsten Schritt wird nun die Schalterstufe 1 geroutet.

Listing A.12: Routing der Schalterstufe 1

```
 -- Stufe 1
  -- Bestimmung von S
 s1_S1 <= '1' when ((s1_I2(1) = s1_I0(1)) and
             (s1_I2(0) = '1') and (s1_I0(0) = '1')) or
                   ((s1_I3(1) = s1_I1(1)) and
             (s1_I3(0) = '1') and (s1_I1(0) = '1')) else '0';
 s1_S3 <= '1' when ((s1_I6(1) = s1_I4(1)) and
             (s1_I6(0) = '1') and (s1_I4(0) = '1')) or
                   ((s1_I7(1) = s1_I5(1)) and
             (s1_I7(0) = '1') and (s1_I5(0) = '1')) else '0';
```

```
-- Schalterausgänge setzen
s1_I0_1 <= s1_I0;
s1_I1_1 <= s1_I1;
s1_I2_1 <= s1_I2 when s1_S1 = '0' else s1_I3;
s1_I3_1 <= s1_I2 when s1_S1 = '1' else s1_I3;
s1_I4_1 <= s1_I4;
s1_I5_1 <= s1_I5;
s1_I6_1 <= s1_I6 when s1_S1 = '0' else s1_I7;
s1_I7_1 <= s1_I6 when s1_S1 = '1' else s1_I7;
-- Verdrahtung Stufe 1 => Stufe 2 (Unshuffle Permutation)
s2_I0 <= s1_I0_1;
s2_I1 <= s1_I2_1;
s2_I2 <= s1_I1_1;
s2_I3 <= s1_I3_1;
s2_I4 <= s1_I4_1;
s2_I5 <= s1_I6_1;
s2_I6 <= s1_I5_1;
s2_I7 <= s1_I7_1;
```

Mit der Schalterstufe 1 ist nun das Routing im Separationsteil des Netzes abgeschlossen. Im nächsten Schritt folgt das Routing im Approximationsteil, beginnend mit der Schalterstufe 2.

Listing A.13: Routing der Schalterstufe 2

```
-- Stufe 2
 -- Bestimmung von S
 s2_S0 <= '1' when ((s2_I0(1) = '1') and (s2_I0(0) = '1')) or
                   ((s2_I1(1) = '0') and (s2_I1(0) = '1')) else '0';
 s2_S1 <= '1' when ((s2_I2(1) = '1') and (s2_I2(0) = '1')) or
                   ((s2_I3(1) = '0') and (s2_I3(0) = '1')) else '0';
 s2_S2 <= '1' when ((s2_I4(1) = '1') and (s2_I4(0) = '1')) or
                   ((s2_I5(1) = '0') and (s2_I5(0) = '1')) else '0';
 s2_S3 <= '1' when ((s2_I6(1) = '1') and (s2_I6(0) = '1')) or
                   ((s2_I7(1) = '0') and (s2_I7(0) = '1')) else '0';
 -- Schalterausgänge setzen
 s2_I0_1 <= s2_I0 when s2_S0 = '0' else s2_I1;
 s2_I1_1 <= s2_I0 when s2_S0 = '1' else s2_I1;
 s2_I2_1 <= s2_I2 when s2_S1 = '0' else s2_I3;
 s2_I3_1 <= s2_I2 when s2_S1 = '1' else s2_I3;
 s2_I4_1 <= s2_I4 when s2_S2 = '0' else s2_I5;
 s2_I5_1 <= s2_I4 when s2_S2 = '1' else s2_I5;
 s2_I6_1 <= s2_I6 when s2_S3 = '0' else s2_I7;
 s2_I7_1 <= s2_I6 when s2_S3 = '1' else s2_I7;
 -- Verdrahtung Stufe 2 => Stufe 3 (Shuffle Permutation)
 s3_I0 <= s2_I0_1;
```

```
  s3_I1 <= s2_I2_1;
  s3_I2 <= s2_I1_1;
  s3_I3 <= s2_I3_1;
  s3_I4 <= s2_I4_1;
  s3_I5 <= s2_I6_1;
  s3_I6 <= s2_I5_1;
  s3_I7 <= s2_I7_1;
```

Das Routing der Schalterstufe 2 ist damit abgeschlossen. Im nächsten Schritt folgt nun das Routing der Schalterstufe 3.

Listing A.14: Routing der Schalterstufe 3

```
-- Stufe 3
  -- Bestimmung von S
  s3_S0 <= '1' when ((s3_I0(2) = '1') and (s3_I0(0) = '1')) or
                    ((s3_I1(2) = '0') and (s3_I1(0) = '1')) else '0';
  s3_S1 <= '1' when ((s3_I2(2) = '1') and (s3_I2(0) = '1')) or
                    ((s3_I3(2) = '0') and (s3_I3(0) = '1')) else '0';
  s3_S2 <= '1' when ((s3_I4(2) = '1') and (s3_I4(0) = '1')) or
                    ((s3_I5(2) = '0') and (s3_I5(0) = '1')) else '0';
  s3_S3 <= '1' when ((s3_I6(2) = '1') and (s3_I6(0) = '1')) or
                    ((s3_I7(2) = '0') and (s3_I7(0) = '1')) else '0';
  -- Schalterausgänge setzen
  s3_I0_1 <= s3_I0 when s3_S0 = '0' else s3_I1;
  s3_I1_1 <= s3_I0 when s3_S0 = '1' else s3_I1;
  s3_I2_1 <= s3_I2 when s3_S1 = '0' else s3_I3;
  s3_I3_1 <= s3_I2 when s3_S1 = '1' else s3_I3;
  s3_I4_1 <= s3_I4 when s3_S2 = '0' else s3_I5;
  s3_I5_1 <= s3_I4 when s3_S2 = '1' else s3_I5;
  s3_I6_1 <= s3_I6 when s3_S3 = '0' else s3_I7;
  s3_I7_1 <= s3_I6 when s3_S3 = '1' else s3_I7;
  -- Verdrahtung Stufe 3 => Stufe 4 (Shuffle Permutation)
  s4_I0 <= s3_I0_1;
  s4_I1 <= s3_I2_1;
  s4_I2 <= s3_I1_1;
  s4_I3 <= s3_I3_1;
  s4_I4 <= s3_I4_1;
  s4_I5 <= s3_I6_1;
  s4_I6 <= s3_I5_1;
  s4_I7 <= s3_I7_1;
```

Das Routing der Schalterstufe 3 ist damit abgeschlossen. Im nächsten Schritt folgt nun das Routing der Schalterstufe 4.

Listing A.15: Routing der Schalterstufe 4

```
-- Stufe 4
  -- Bestimmung von S
  s4_S0 <= '1' when ((s4_I0(3) = '1') and (s4_I0(0) = '1')) or
                    ((s4_I1(3) = '0') and (s4_I1(0) = '1')) else '0';

  s4_S1 <= '1' when ((s4_I2(3) = '1') and (s4_I2(0) = '1')) or
                    ((s4_I3(3) = '0') and (s4_I3(0) = '1')) else '0';

  s4_S2 <= '1' when ((s4_I4(3) = '1') and (s4_I4(0) = '1')) or
                    ((s4_I5(3) = '0') and (s4_I5(0) = '1')) else '0';

  s4_S3 <= '1' when ((s4_I6(3) = '1') and (s4_I6(0) = '1')) or
                    ((s4_I7(3) = '0') and (s4_I7(0) = '1')) else '0';
-- Ende Routing-Algorithmus
```

Mit dem Routing der Schalterstufe 4 ist nun auch das Routing des Approximationsteils abgeschlossen. Damit sind alle Schalterstellungen im Netz bestimmt. Anhand dieser Schalterstellungen können nun die Verbindungswünsche im Netz vermittelt werden, wie der nachfolgende Abschnitt zeigt.

A.4.7 Datenübertragung

Leitungsvermittlung. Nachdem im Abschnitt A.4.6 Routing, S. 261 ff. die Schalterstellungen ermittelt wurden, können nun die Verbindungswünsche im Netz vermittelt werden. Die Skalierbarkeit des Netzwerks beruht neben dem Routing-Verfahren auf der leitungsvermittelnden Funktion des Netzes[16], d.h. es wird über

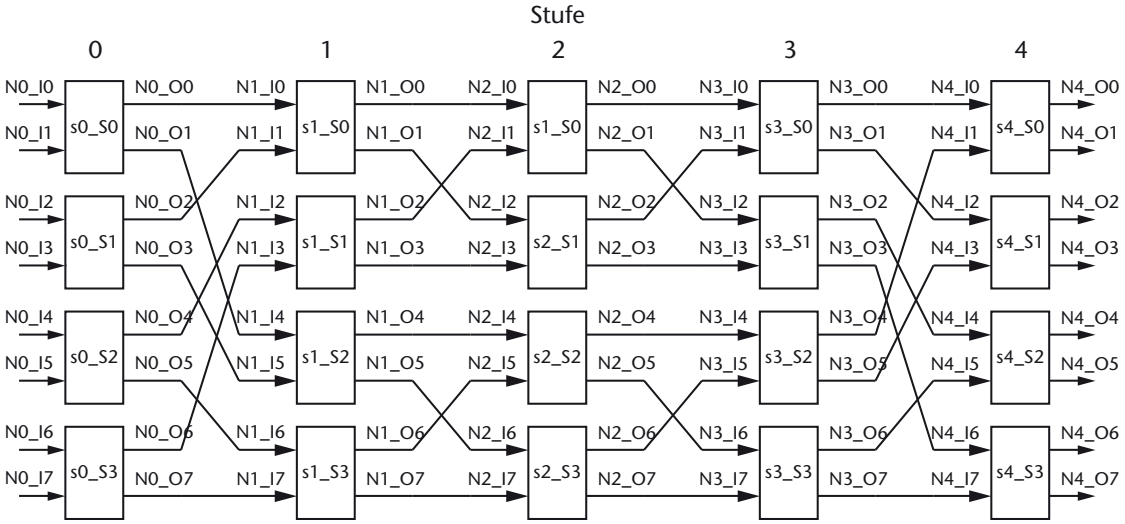


Abbildung A.9: Bezeichnungen für die Leitungsvermittlung im Beneš-Netz

[16] Funktionsweise der Leitungsvermittlung in FPGA-Hardware beschrieben in 5.5.6.1 Verdrahtungsstufen, S. 163 und 5.5.6.2 Kreuzschalter, S. 163 f.

Schalter- und Verdrahtungsstufen eine direkte Verbindung vom Sender zum Empfänger aufgebaut. Zum besseren Verständnis sind die nachfolgend verwendeten Bezeichnungen in der **Abbildung A.9** noch einmal im Netz dargestellt. Zunächst werden die benötigten Signale, d.h. die Schalterstellungen im Netz als einzelne Bits (aus Routingalgorithmus) und die zu vermittelnden Leitungen im Netz als Bit-Vektor, deklariert. Zusätzlich wird für jeden MicroBlaze Softprozessor ein FSL-Interface zum Netzwerk deklariert.

Listing A.16: Deklaration der benötigten Signale

```
entity min is
 port(
  -- FSL_Interface für MicroBlaze 0
  ...
  MB0_FSL_S_Clk : out std_logic;  -- Clock für asynchrone Kommunikation
  MB0_FSL_S_Read : out std_logic;  -- Daten gelesen
  MB0_FSL_S_Data : in std_logic_vector(0 to 31); -- Empfängerdaten
  MB0_FSL_S_Control : in std_logic;  -- Steuerdaten (Zieladresse)
  MB0_FSL_S_Exists : in std_logic;  -- Daten vorhanden
  MB0_FSL_M_Clk : out std_logic;  -- Clock für asynchrone Kommunikation
  MB0_FSL_M_Write : out std_logic;  -- Daten schreiben
  MB0_FSL_M_Data : out std_logic_vector(0 to 31); -- Senderdaten
  ...
  -- FSL_Interface für MicroBlaze 1
  ...
 );
architecture NETWORK of min is
-- Deklaration: Anschlüsse Kreuzschalter
-- Bezeichner: N: Netzanschluss
-- Belegung: 0..31: Data, 32: Exists, 33: Control
signal N0_I0: std_logic_vector(0 to 34);
signal N0_I1: std_logic_vector(0 to 34);
signal N0_I2: std_logic_vector(0 to 34);
signal N0_I3: std_logic_vector(0 to 34);
  ...
signal N4_O4: std_logic_vector(0 to 34);
signal N4_O5: std_logic_vector(0 to 34);
signal N4_O6: std_logic_vector(0 to 34);
signal N4_O7: std_logic_vector(0 to 34);
...
-- Deklaration: Status Kreuzschalter
-- Bezeichner: s: Stufe, S: Schalter
signal s0_S0, s0_S1, s0_S2, s0_S3 : bit; -- Stufe 0
```

```
signal s1_S0, s1_S1, s1_S2, s1_S3 : bit; -- Stufe 1
signal s2_S0, s2_S1, s2_S2, s2_S3 : bit; -- Stufe 2
signal s3_S0, s3_S1, s3_S2, s3_S3 : bit; -- Stufe 3
signal s4_S0, s4_S1, s4_S2, s4_S3 : bit; -- Stufe 4
...
```

Die Leitungsvermittlung im Netz erfolgt von links nach rechts durch die Verkettung von Signalen. Zunächst werden die Signale der Senderseite in das Netzwerk eingespeist.

Listing A.17: Einspeisung der Senderseite

```
begin
...
-- Zuweisung der FSL-Eingänge
N0_I0(0 to 31) <= MB0_FSL_S_Data;
N0_I1(0 to 31) <= MB1_FSL_S_Data;
N0_I2(0 to 31) <= MB2_FSL_S_Data;
...
N0_I0(32) <= MB0_FSL_S_Exists;
N0_I1(32) <= MB1_FSL_S_Exists;
N0_I2(32) <= MB2_FSL_S_Exists;
...
N0_I0(33) <= MB0_FSL_S_Control;
N0_I1(33) <= MB1_FSL_S_Control;
N0_I2(33) <= MB2_FSL_S_Control;
...
N0_I7(33) <= MB7_FSL_S_Control;
```

Nun werden die Signale im Wechsel aus Kreuzschalterstufen und Verdrahtungsstufen durch das Netz verkettet.

Listing A.18: Verkettete Signale im Netz

```
-- Kreuzschalter Stufe 0
N0_O0 <= N0_I0 when s0_S0 = '0' else N0_I1;
N0_O1 <= N0_I0 when s0_S0 = '1' else N0_I1;
N0_O2 <= N0_I2 when s0_S1 = '0' else N0_I3;
N0_O3 <= N0_I2 when s0_S1 = '1' else N0_I3;
N0_O4 <= N0_I4 when s0_S2 = '0' else N0_I5;
N0_O5 <= N0_I4 when s0_S2 = '1' else N0_I5;
N0_O6 <= N0_I6 when s0_S3 = '0' else N0_I7;
N0_O7 <= N0_I6 when s0_S3 = '1' else N0_I7;
-- Verdrahtung von Stufe 0 auf Stufe 1
N1_I0 <= N0_O0;
```

```
N1_I1 <= N0_O2;
N1_I2 <= N0_O4;
N1_I3 <= N0_O6;
N1_I4 <= N0_O1;
N1_I5 <= N0_O3;
N1_I6 <= N0_O5;
N1_I7 <= N0_O7;
-- Kreuzschalter Stufe 1
N1_O0 <= N1_I0 when s1_S0 = '0' else N1_I1;
N1_O1 <= N1_I0 when s1_S0 = '1' else N1_I1;
N1_O2 <= N1_I2 when s1_S1 = '0' else N1_I3;
N1_O3 <= N1_I2 when s1_S1 = '1' else N1_I3;
N1_O4 <= N1_I4 when s1_S2 = '0' else N1_I5;
N1_O5 <= N1_I4 when s1_S2 = '1' else N1_I5;
N1_O6 <= N1_I6 when s1_S3 = '0' else N1_I7;
N1_O7 <= N1_I6 when s1_S3 = '1' else N1_I7;
-- Verdrahtung von Stufe 1 auf Stufe 2
N2_I0 <= N1_O0;
N2_I1 <= N1_O2;
N2_I2 <= N1_O1;
N2_I3 <= N1_O3;
N2_I4 <= N1_O4;
N2_I5 <= N1_O6;
N2_I6 <= N1_O5;
N2_I7 <= N1_O7;
-- Kreuzschalter Stufe 2
N2_O0 <= N2_I0 when s2_S0 = '0' else N2_I1;
N2_O1 <= N2_I0 when s2_S0 = '1' else N2_I1;
N2_O2 <= N2_I2 when s2_S1 = '0' else N2_I3;
N2_O3 <= N2_I2 when s2_S1 = '1' else N2_I3;
N2_O4 <= N2_I4 when s2_S2 = '0' else N2_I5;
N2_O5 <= N2_I4 when s2_S2 = '1' else N2_I5;
N2_O6 <= N2_I6 when s2_S3 = '0' else N2_I7;
N2_O7 <= N2_I6 when s2_S3 = '1' else N2_I7;
-- Verdrahtung von Stufe 2 auf Stufe 3
N3_I0 <= N2_O0;
N3_I1 <= N2_O2;
N3_I2 <= N2_O1;
N3_I3 <= N2_O3;
N3_I4 <= N2_O4;
N3_I5 <= N2_O6;
N3_I6 <= N2_O5;
N3_I7 <= N2_O7;
```

```
-- Kreuzschalter Stufe 3
N3_O0 <= N3_I0 when s3_S0 = '0' else N3_I1;
N3_O1 <= N3_I0 when s3_S0 = '1' else N3_I1;
N3_O2 <= N3_I2 when s3_S1 = '0' else N3_I3;
N3_O3 <= N3_I2 when s3_S1 = '1' else N3_I3;
N3_O4 <= N3_I4 when s3_S2 = '0' else N3_I5;
N3_O5 <= N3_I4 when s3_S2 = '1' else N3_I5;
N3_O6 <= N3_I6 when s3_S3 = '0' else N3_I7;
N3_O7 <= N3_I6 when s3_S3 = '1' else N3_I7;
-- Verdrahtung von Stufe 3 auf Stufe 4
N4_I0 <= N3_O0;
N4_I1 <= N3_O4;
N4_I2 <= N3_O1;
N4_I3 <= N3_O5;
N4_I4 <= N3_O2;
N4_I5 <= N3_O6;
N4_I6 <= N3_O3;
N4_I7 <= N3_O7;
-- Kreuzschalter Stufe 4
N4_O0 <= N4_I0 when s4_S0 = '0' else N4_I1;
N4_O1 <= N4_I0 when s4_S0 = '1' else N4_I1;
N4_O2 <= N4_I2 when s4_S1 = '0' else N4_I3;
N4_O3 <= N4_I2 when s4_S1 = '1' else N4_I3;
N4_O4 <= N4_I4 when s4_S2 = '0' else N4_I5;
N4_O5 <= N4_I4 when s4_S2 = '1' else N4_I5;
N4_O6 <= N4_I6 when s4_S3 = '0' else N4_I7;
N4_O7 <= N4_I6 when s4_S3 = '1' else N4_I7;
```

Zum Abschluss werden die Ausgänge des Netzes auf die FSL-Interfaces der Softprozessoren geschrieben. Außerdem wird der Empfang der Daten beim Sender quittiert, wenn die Leitung vermittelt ist (= Validierungsbit gesetzt).

Listing A.19: Schreiben der Ausgänge

```
-- Zuweisung der FSL-Ausgänge
MB0_FSL_M_Data <= N4_O0(0 to 31);
MB1_FSL_M_Data <= N4_O1(0 to 31);
MB2_FSL_M_Data <= N4_O2(0 to 31);
...
MB0_FSL_M_Write <= N4_O0(32) when N4_O0(33) = '0' else '0';
MB1_FSL_M_Write <= N4_O1(32) when N4_O1(33) = '0' else '0';
MB2_FSL_M_Write <= N4_O2(32) when N4_O2(33) = '0' else '0';
...
MB0_FSL_S_Read <= MB0_FSL_S_Exists when s0_I0(0) = '1' else '0';
```

```
MB1_FSL_S_Read <= MB1_FSL_S_Exists when s0_I1(0) = '1' else '0';
MB2_FSL_S_Read <= MB2_FSL_S_Exists when s0_I2(0) = '1' else '0';
...
MB7_FSL_S_Read <= MB7_FSL_S_Exists when s0_I7(0) = '1' else '0';
```

Damit ist die Datenübertragung im Beispiel-Netz abgeschlossen. Damit die Verbindungen geroutet werden können, müssen abschließend noch die Zieladressen der Sender an den Routing-Algorithmus übergeben werden[17].

Listing A.20: Übergabe der Zieladressen ohne Arbitrierung

```
-- Zieladressen in das Netzwerk speisen
The_circuit_switching : process (FSL_Clk) is

  type xyz is array (0 to 7) of integer range 0 to 15;
  variable Adresse: xyz;

begin  -- process The_circuit_switching

  if FSL_Clk'event and FSL_Clk = '1' then  -- Rising clock edge

    if FSL_Rst = '1' then  -- Synchronous reset (active high)
      -- Zieladressen der Sender initialisieren
      Adresse(0) := 0;
      Adresse(1) := 1;
      ...
      Adresse(7) := 7;
    end if;

  -- Zieladressen der Sender übernehmen (wenn FSL_S_Control = '1')
  -- MicroBlaze 0
  if MB0_FSL_S_Exists = '1' and MB0_FSL_S_Control = '1' then
    -- Nummer des Zielprozessors
    Adresse(0) := conv_integer(MB0_FSL_S_Data(28 to 31));
  end if;
  -- MicroBlaze 1
  if MB1_FSL_S_Exists = '1' and MB1_FSL_S_Control = '1' then
    -- Nummer des Zielprozessors
    Adresse(1) := conv_integer(MB1_FSL_S_Data(28 to 31));
  end if;
...
  -- MicroBlaze 7
```

[17] vgl. A.4.2 Verbindungsaufbau, S. 257 f.

```
  if MB7_FSL_S_Exists = '1' and MB7_FSL_S_Control = '1' then
    -- Nummer des Zielprozessors
    Adresse(7) := conv_integer(MB7_FSL_S_Data(28 to 31));
  end if;

  -- Zieladressen in die erste Schalterstufe speisen
  s0_I0 <= To_bitvector(conv_std_logic_vector(Adresse(0),4));
  s0_I1 <= To_bitvector(conv_std_logic_vector(Adresse(1),4));
...
  s0_I7 <= To_bitvector(conv_std_logic_vector(Adresse(7),4));

end if;

end process The_circuit_switching;

end architecture NETWORK
```

A.5 Energiemanagement

A.5.1 Variable Steuerung des Prozessortaktes

Clock Rate Controller. Die Verlustleistung des Echtzeitparallelrechners soll durch den Einsatz einer variablen Steuerung des Prozessortaktes für jeden einzelnen Prozessor zur Laufzeit optimiert werden können[18]. Wie in der **Abbildung A.10** dargestellt, wird für jeden Prozessor zusätzlich ein *Clock Rate Controller* eingeführt, mit dessen Hilfe die Taktfrequenz des Prozessors gesteuert werden kann.

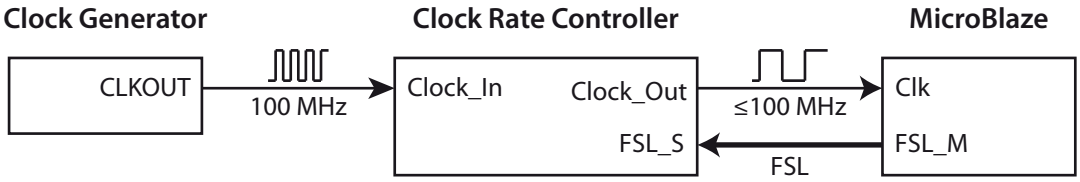


Abbildung A.10: Aufbau der variablen Prozessortaktsteuerung im FPGA

Variable Clock Rate. Damit die Taktfrequenz des Prozessors zur Laufzeit vom Programm verändert werden kann, muss der Teilerfaktor vom MicroBlaze an den Clock Rate Controller übergeben werden. Dies geschieht über eine unidirektionale FSL-Verbindung (vgl. Abbildung A.10).

[18] vgl. 6.3.3 Dynamische Anpassung der Prozessorleistung, S. 190 ff.

Clock Gating. Der prototypisch implementierte Clock Rate Controller arbeitet als Taktteiler auf Basis des Gated-Clock-Prinzips. Die Implementierung des Clock Rate Controllers erfolgt als IP-Core, dessen Interface in der Abbildung A.10 dargestellt ist. Der komplette VHDL-Code des IP-Cores ist im **Listing A.21** aufgeführt.

Listing A.21: Clock Rate Controller (IP-Core)

```
-- clock rate controller --
-- author: Stefan Aust --
-- IP core: clk_ctrl.vhd --

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity clk_ctrl is
   port(
      -- FSL interface to softprocessor
      FSL_Clk     : in  std_logic;
      FSL_Rst     : in  std_logic;
      FSL_S_Clk   : out std_logic;
      FSL_S_Read  : out std_logic;
      FSL_S_Data  : in  std_logic_vector(0 to 31);
      FSL_S_Control     : in  std_logic;
      FSL_S_Exists      : in  std_logic;
      -- clock signal interface
      Clock_In : in std_logic;
      Clock_Out : out std_logic);

end clk_ctrl;

architecture ClockRateController of clk_ctrl is
   -- declaration of signals
   signal clockCount, clockCounter : integer range 0 to 255 := 0;
   signal clockEnable : std_logic := '0';

begin
   -- generate gated clock signal
   GatedClock : process (Clock_In) is
   begin
      if (clockEnable = '1') then
         Clock_Out <= Clock_In;   -- generate Clock_Out signal
```

```
      end if;
      if rising_edge(Clock_In) then
         if (clockCounter < clockCount) then
            clockCounter <= clockCounter + 1;    -- clock counter inc
            clockEnable <= '0';
         else
            clockCounter <= 0;    -- clock counter reset
            clockEnable <= '1';
         end if;
      end if;
   end process GatedClock;

   -- get clockrate via FSL communication from softprocessor
   GetClockRate : process (FSL_Clk) is
   begin
      if FSL_Clk'event and FSL_Clk = '1' then    -- rising clock edge?
         if (FSL_S_Exists = '1') then    -- new clock rate from MB?
            -- read factor from softprocessor (8 bit value)
            clockCount <= conv_integer(FSL_S_Data(24 to 31));
            FSL_S_Read <= '1';    -- set FSL acknowledge
         else
            FSL_S_Read <= '0';    -- reset FSL acknowledge
         end if;
      end if;
   end process GetClockRate;
end architecture ClockRateController;
```

Übergabe der Taktrate. Je nach erforderlicher Rechenleistung wird die Taktrate des Prozessors vom Programm zur Laufzeit angepasst. Hierfür muss im Programm die im **Listing A.22** aufgeführte Funktion *setClockRate* mit der Angabe des Taktteilers aufgerufen werden. Bei Übergabe einer „0“ würde der Prozessor mit 100 MHz getaktet werden, was der vollen Rechenleistung entspricht. Der maximale Teilerfaktor ist aus praktischen Gründen im Prototyp auf 8 Bit begrenzt, was einer realen Prozessortaktrate von 390 kHz entspricht. Die dynamische Verlustleistung des Prozessors kann dadurch auf bis zu 0,4 % gegenüber der Verlustleisung bei maximaler Rechenleistung gesenkt werden[19]. Aufgrund des 32-Bit breiten FSL-Datenkanals könnte der Taktteiler theoretisch bis auf 2^{32} erweitert werden, allerdings macht es in praxi wenig Sinn, den Prozessor mit 0,02 Hz zu takten.

Listing A.22: Taktrate setzen (MicroBlaze)

```
#include "mb_interface.h"
```

[19] vgl. 6.2.1 Einfluss der Taktfrequenz, S. 178 ff.

```
// set new clockrate
void setClockRate (int factor){
   if (factor < 0) return;   // check lower limit
   if (factor > 255) return;   // check upper limit
   // write factor via FSL to gated clock
   microblaze_bwrite_datafsl (factor,0);
   return;
}
```

A.5.2 Konfiguration der Prozessorarchitektur

Konfiguration von Softprozessoren. Die Architektur von Softprozessoren wird, wie andere Komponenten im FPGA auch, durch VHDL beschrieben. Aus diesem Grund stellt die Architektur von Softprozessoren keine feste Größe dar, sondern kann im Rahmen eines Software-First-Designs[20] applikationsabhängig konfiguriert werden. Da jede zusätzliche Komponente im Softprozessor neben zusätzlichen FPGA-Ressourcen auch zusätzliche Energie verbraucht[21], macht es aus energetischer Sicht Sinn, die Architektur jedes einzelnen Softprozessors im MPSoC auf die unbedingt erforderlichen Komponenten zu beschränken. Die Konfiguration der Softprozessoren wird bei Xilinx im Projektverzeichnis in der Datei *system.mhs*[22] abgelegt. Im **Listing A.23** sind die Konfigurationsdaten für einen MicroBlaze Softprozessor der Fa. Xilinx dargestellt. An dieser Stelle können Prozessorkomponenten wie z.B. Fließkommaeinheit oder Multiplizierer einzeln an- und abgewählt werden.

Listing A.23: Konfiguration eines MicroBlaze Softprozessors

```
-- configuration file: system.mhs
BEGIN microblaze
   ...
   PARAMETER C_AREA_OPTIMIZED = 1    -- 3-stufige Pipeline
   PARAMETER C_USE_FPU = 1(2)        -- Fließkommaeinheit Basic(Ext)
   PARAMETER C_USE_BARREL = 1        -- Barrel Shifter
   PARAMETER C_USE_DIV = 1           -- Integer Divider
   PARAMETER C_USE_HW_MUL = 1(2)     -- Multiplizierer 32(64) Bit
   ...
END
```

[20] vgl. 4.5.1 System-Design, S. 60 ff.

[21] vgl. 6.2.6 Einfluss des Prozessordesigns, S. 186 f.

[22] Microprocessor Hardware Specification, vgl. [Xi11a, S. 17 ff.]

Anhang B - Leistungsmessungen an FPGAs

B.1 Grundlagen

Die FPGA-Boards. Für die Messungen stehen Entwicklungsboards mit 3 unterschiedlichen FPGAs zur Verfügung (**Tabelle B.1**). Alle verwendeten Boards besitzen ein externes Netzteil zur Stromversorgung. In diese externe Versorgung werden zwei Messgeräte für Strom und Spannung geschaltet, mit deren Hilfe sich die Leistungsaufnahme der Boards ermitteln lässt.

Tabelle B.1: Liste der untersuchten FPGAs

FPGA	Typ	Board	Hersteller
Xilinx Spartan-3	XC3S1000	Starter Kit Board	Digilent
Xilinx Virtex-4	XC4VFX100	XpressFX	PLDA
Xilinx Virtex-5	XC5VLX50T	ML505	Xilinx

Die Messgeräte. Für die Messungen werden zwei Multimeter gleichen Typs verwendet (**Tabelle B.2**). Die Genauigkeit der Messgeräte besitzt bei den Messungen keine signifikante Bedeutung, da die Varianz in der Stromaufnahme aufgrund der unterschiedlichen Schaltungssynthesen der Testumgebungen und des Einflusses weiterer Komponenten auf den Boards sehr viel größer ausfallen. Auf eine Angabe der Messgenauigkeit wird daher in den Messungen verzichtet.

Tabelle B.2: Liste der verwendeten Messgeräte

Messgröße	Gerät	Hersteller	Typ	Messbereich	Genauigkeit
Spannung	Scope Meter	Voltcraft	DMM 750	4 V=	+/- (0,3% + 5digits)
Strom	Scope Meter	Voltcraft	DMM 750	400 mA=	+/- (1,2% + 10digits)
				4 A=	+/- (1,5% + 10digits)

Der Messaufbau. Der Messaufbau erfolgt mit einer regelbaren Spannungsquelle für die Stromversorgung anstelle des externen Netzteils des Board-Herstellers und zwei Messgeräten (**Abbildung B.1**). Ein Messgerät wird als Amperemeter und das

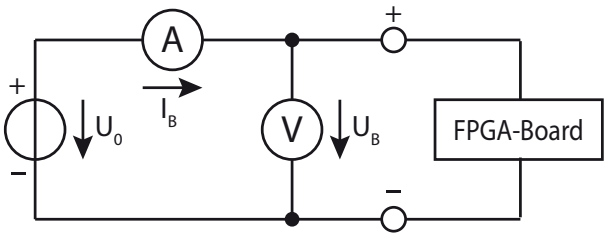


Abbildung B.1: Messaufbau

andere wird als Voltmeter verwendet. Beide Messgeräte sind handelsübliche Multimeter mit geeigneten Messbereichen (vgl. Tabelle B.2).

Leistungsmessung. Die Leistungsmessung am FPGA-Board erfolgt in spannungsrichtiger Messung. Dies hat zwei Gründe. Erstens besitzt das Amperemeter in den kleinen Messbereichen einen relativ hohen Innenwiderstand. Durch den daraus resultierenden hohen Spannungsabfall am Messgerät besteht die Gefahr, dass die berechnete Verlustleistung stark verfälscht wird. Zweitens kann, je nach Spannungsregler auf dem Board, die Stromaufnahme I_B von der Eingangsspannung U_B am Board selbst abhängen. Da die Spannung U_B wiederum vom Spannungsabfall über das Amperemeter (bzw. von der Stromaufnahme des Boards) abhängt, muss die Spannungsquelle U_0 gegebenenfalls für einen konstanten Wert von U_B am Board nachjustiert werden können. Die vom FPGA-Board aufgenommene Leistung P ergibt sich mit:

$$P = U_B \cdot I_B \tag{B.1}$$

Da für die Messungen handelsübliche Entwicklungsboards verwendet werden, ist es leider nicht möglich, den Leistungsbedarf direkt am FPGA zu messen. Die Bedeutung der board-internen Spannungsregler bei der Bewertung der Messdaten lässt daher Raum für Diskussionen. Aus zwei Gründen halte ich die Messdaten trotzdem für verwertbar. Erstens geht es bei den Messergebnissen nicht primär um absolute Werte, sondern um den Nachweis bestimmter Trends und damit um die Relation der Messwerte zueinander. Zweitens ist der Einfluss von potentiellen energiesparenden Maßnahmen in der Multiprozessorumgebung des FPGAs auf den Leistungsbedarf der gesamten Platine bzw. des fertigen Steuergerätes entscheidend für deren Bewertung.

Die Testumgebungen. Für die Messungen wurden verschiedene Multiprozessorsysteme auf VHDL-Basis generiert. Die verwendeten Softprozessoren sind 32-Bit

RISC-Prozessoren der Firma Xilinx vom Typ MicroBlaze. Für die Messungen wurden Default-Einstellungen verwendet, d.h. es gibt keine zusätzlichen Rechenwerke oder Caches. Sowohl die Hardware als auch die Software der Testumgebungen wurden im Xilinx EDK erstellt. Die Softprozessoren besitzen einen vorgeschalteten Taktteiler (**Abbildung B.2**), über den der Systemtakt des Taktgenerators für die Messung der dynamischen Verlustleistung in unterschiedliche Taktraten geteilt werden kann. Der Faktor kann von außen über DIP-Schalter vorgegeben werden. Für die Messung der statischen Verlustleistung besteht zusätzlich zur Taktteilung die Möglichkeit, den Takt über die DIP-Schalter komplett zu unterbrechen.

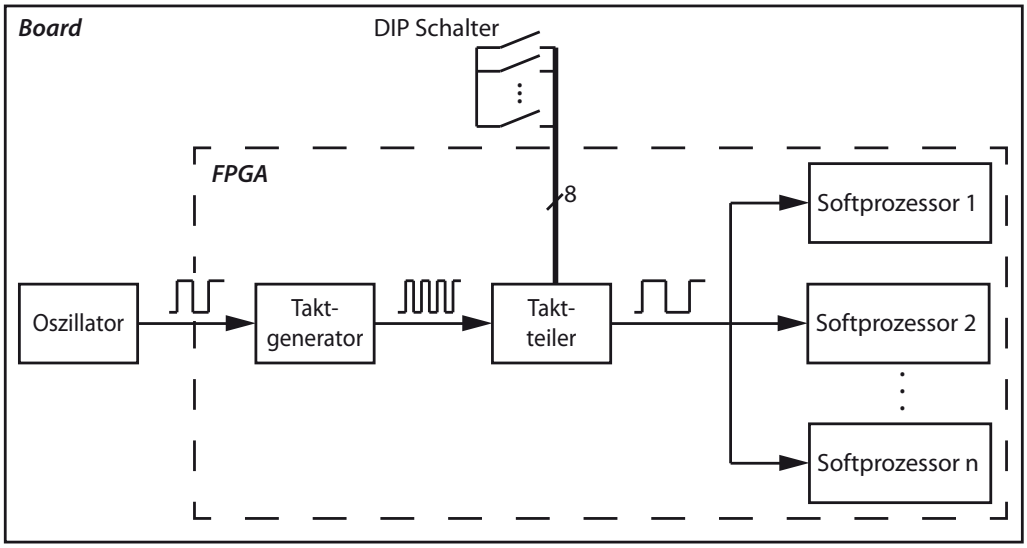


Abbildung B.2: Testumgebung für die Messungen am Multiprozessorsystem

Hinweis: Es hat sich bei den Messungen gezeigt, dass bei einer Änderung in der Architektur des Multiprozessorsystems, z.B. durch Hinzufügen eines weiteren Softprozessors, alle früher generierten Daten gelöscht und neu synthetisiert werden sollten. Anderenfalls kann es in den Messungen zu größeren Abweichungen kommen.

B.2 Verlustleistung in Abhängigkeit von der Taktfrequenz

Inhalt. Die Messung soll klären, ob und wie die dynamische Verlustleistung eines Multiprozessorsystems von der Taktfrequenz abhängt, mit der die Prozessoren getaktet werden.

Aufnahme der Messdaten. Die Messdaten in diesem Abschnitt wurden am Spartan-3 Board und am Virtex-5 Board aufgenommen. Für die Messungen am

Virtex-5 Board wurden Testumgebungen mit 1 bis 8 Softprozessoren implementiert. Die Messungen am Spartan-3 Board beschränken sich aufgrund der deutlich geringeren Anzahl der Logikgatter auf Testumgebungen mit 1 bis 6 Softprozessoren. Die Größe der Daten- und Befehlsspeicher je Softprozessor beträgt bei beiden FPGAs je 8 KiB. Dadurch stehen jedem Softprozessor insgesamt 16 KiB Speicher zur Verfügung, die mit den FPGA-internen Block-RAMs realisiert sind. Zusätzlich wurde ein Taktteiler (*clock divider*) für den Systemtakt implementiert, mit dem während den Messungen verschiedene Taktraten von außen über DIP-Switches eingestellt werden können, ohne für jeden Messwert neu synthetisieren zu müssen.

Tabelle B.3: Messdaten aus Messung S3-110216

Anzahl der Prozessoren	**1 P**	**2 P**	**3 P**	**4 P**	**5 P**	**6 P**
f_{takt} **[MHz]**	**Strom** I_b **[mA]**					
0	157,4	157,4	157,7	158,1	159,2	159,5
0,75	160,4	160,9	161,7	162,5	163,4	164,2
3,75	161,6	164,4	167,4	170,1	173,2	176,2
7,5	163,3	168,9	174,6	179,9	185,7	191,4
12,5	166,7	175,9	185,2	193,6	203	212,1
15	167,5	178,7	189,9	200,3	211,6	222,8
18,75	170,8	184,6	198,6	211,5	225,4	239,3
25	174,1	192,6	210,5	227,4	247,5	266,2
37,5	182,5	210,6	238,5	264,3	292,6	320,1
75	206,4	262,6	317,8	368,8	427	484

Tabelle B.4: Messdaten aus Messung V5-110216

Anzahl der Prozessoren	**1 P**	**2 P**	**3 P**	**4 P**	**5 P**	**6 P**	**7 P**	**8 P**
f_{takt} **[MHz]**	**Strom** I_b **[mA]**							
0	1,094	1,094	1,094	1,093	1,094	1,094	1,093	1,093
1	1,094	1,095	1,095	1,094	1,096	1,096	1,094	1,095
5	1,095	1,096	1,096	1,096	1,098	1,098	1,097	1,098
10	1,096	1,097	1,098	1,098	1,100	1,101	1,101	1,102
25	1,098	1,100	1,105	1,105	1,110	1,111	1,113	1,115
50	1,101	1,106	1,115	1,115	1,124	1,126	1,130	1,135
100	1,108	1,117	1,128	1,136	1,152	1,158	1,164	1,173

Die Stromaufnahme wurde am Anschluss für das externe Netzteil gemessen. Beide FPGA-Boards werden von extern mit einer Gleichspannung von $U_B = 5\ V$ versorgt [Xi05, Xi08a]. Die in den Messungen ermittelten Messdaten sind für das Spartan-3 Board in der **Tabelle B.3** und für das Virtex-5 Board in der **Tabelle B.4** dokumentiert.

Hinweis: Bei den Messungen am verwendeten Spartan-3 Board ist darauf zu achten, dass die Versorgungsspannung U_B am Board nicht unter die Schwellspannung des Spannungsreglers von 4,75 V fällt [Na05]. Bei höheren Leistungen muss die Spannung U_0 entsprechend nachkorrigiert werden, da möglicherweise der Spannungsabfall am Amperemeter aufgrund des niedrigen Messbereiches zu hoch ausfällt und so das Messergebnis von I_B verfälscht. Oberhalb von 4,75 V hat die Eingangsspannung U_B keinen Einfluss auf die Stromaufnahme.

Ermittlung der dynamischen Verlustleistung. Die dynamische Verlustleistung P_{dyn} kann nun aus den aufgenommenen Messdaten aus Tabelle B.3 und Tabelle B.4 berechnet werden. Die Verlustleistung ergibt sich aus dem Produkt der Versorgungsspannung $U_B = 5V$ und den gemessenen Stromwerten I_B abzüglich der statischen Grundlast I_{Bstat} bei $f_{takt} = 0MHz$ der jeweiligen Testumgebung.

$$P_{dyn} = \frac{U_B}{I_B - I_{Bstat}} \qquad (B.2)$$

Die **Tabelle B.5** und die **Tabelle B.6** zeigen die ermittelten Werte der dynamischen Verlustleistung P_{dyn} in Abhängigkeit von der Taktfrequenz f_{takt} und der Anzahl der Softprozessoren für das Spartan-3 und das Virtex-5 Board.

Tabelle B.5: Datenanalyse aus Messung S3-110216

Anzahl der Prozessoren	**1 P**	**2 P**	**3 P**	**4 P**	**5 P**	**6 P**
f_{takt} **[MHz]**	**Dynamische Verlustleistung** P_{dyn} **[mW]**					
0,75	15,0	17,5	20,0	22,0	21,0	23,5
3,75	21,0	35,0	48,5	60,0	70,0	83,5
7,5	29,5	57,5	84,5	109,0	132,5	159,5
12,5	46,5	92,5	137,5	177,5	219,0	263,0
15	50,5	106,5	161,0	211,0	262,0	316,5
18,75	67,0	136,0	204,5	267,0	331,0	399,0
25	83,5	176,0	264,0	346,5	441,5	533,5

Anzahl der Prozessoren	**1 P**	**2 P**	**3 P**	**4 P**	**5 P**	**6 P**
f_{takt} **[MHz]**	**Dynamische Verlustleistung** P_{dyn} **[mW]**					
37,5	125,5	266,0	404,0	531,0	667,0	803,0
75	245,0	526,0	800,5	1053,5	1339,0	1622,5

Tabelle B.6: Datenanalyse aus Messung V5-110216

Anzahl der Prozessoren	**1 P**	**2 P**	**3 P**	**4 P**	**5 P**	**6 P**	**7 P**	**8 P**
f_{takt} **[MHz]**	**Dynamische Verlustleistung** P_{dyn} **[mW]**							
1	0	5	5	5	10	10	5	10
5	5	10	10	15	20	20	20	25
10	10	15	20	25	30	35	40	45
25	20	30	45	60	80	85	100	110
50	35	60	85	110	150	160	185	210
100	70	115	170	215	290	320	355	400

Abhängigkeit von der Taktfrequenz. Die Ergebnisse der Messung sind in der **Abbildung B.3** und der **Abbildung B.4** grafisch dargestellt. Sehr gut zu erkennen ist in beiden Abbildungen der lineare Zusammenhang zwischen der dynamischen Verlustleistung P_{dyn} und der Taktfrequenz f_{takt} bei allen getesteten Umgebungen. Lediglich der Abstand zwischen den Steigungen der Geraden variiert ein wenig. Neben den üblichen Messungenauigkeiten durch den Messaufbau und den verwendeten Messgeräten, spielen die Schaltkapazitäten auf dem FPGA eine ganz entscheidende Rolle. Diese Schaltkapazitäten werden durch das Place&Route-Werkzeug bei der Synthese der Hardware auf dem Chip beeinflusst, so dass nach jeder neuen Synthese mit varianten Ergebnissen, zum Beispiel durch veränderte Verbindungskapazitäten, zu rechnen ist. Um ein möglichst genaues Bild über die betrachtete Abhängigkeit der dynamischen Verlustleistung zu bekommen, werden die Messwerte aus Tabelle B.5 und Tabelle B.6 erneut bewertet. Um einen realistischen Wert zu bekommen, werden zunächst die Messungen mit den meisten Softprozessoren herangezogen. Bei diesen Messungen ist das Verhältnis von statischer und dynamischer Verlustleistung besonders günstig, das heißt die Messung erreicht hier die größte Präzision. Da in den Messwerten neben den Softprozessoren noch weitere Komponenten wie Taktteiler, I/Os und Debug-Interface enthalten sind, werden die Messergebnisse aus den Messungen mit nur einem Softprozessor ebenfalls betrachtet. Die Differenz beider Messungen enthält die dynamische

Verlustleistung P_{dyn}, die 5 bzw. 7 Softprozessoren verbrauchen. Die Gleichung B.3 zeigt die Berechnung der dynamischen Verlustleistung P_{dyn} für einen einzelnen Softprozessor.

$$P_{dyn} = \frac{P_{6MB} - P_{1MB}}{5} \tag{B.3}$$

Die durchschnittliche dynamische Verlustleistung P_{dyn} in Abhängigkeit von der Taktfrequenz f_{takt} lässt sich für einen einzelnen Softprozessor mit der Gleichung B.4 beschreiben.

$$P_{dyn}\left(f_{takt}\right) = \frac{P_{6MB} - P_{1MB}}{5 \cdot f_{takt}} \tag{B.4}$$

Die **Abbildung B.5** zeigt die Ergebnisse aus der Gleichung B.4 als durchschnittliche dynamische Verlustleistung P_{dyn} in Abhängigkeit von der Taktfrequenz f_{takt} für einen einzelnen Softprozessor. Anhand der Steigungen der Geraden lassen sich für beide FPGAs die folgenden Abhängigkeiten ablesen:

Spartan-3 $$P_{dyn}\left(f_{takt}\right) \approx 3{,}64 \frac{mW}{MHz} \tag{B.5}$$

Virtex-5 $$P_{dyn}\left(f_{takt}\right) \approx 0{,}48 \frac{mW}{MHz} \tag{B.6}$$

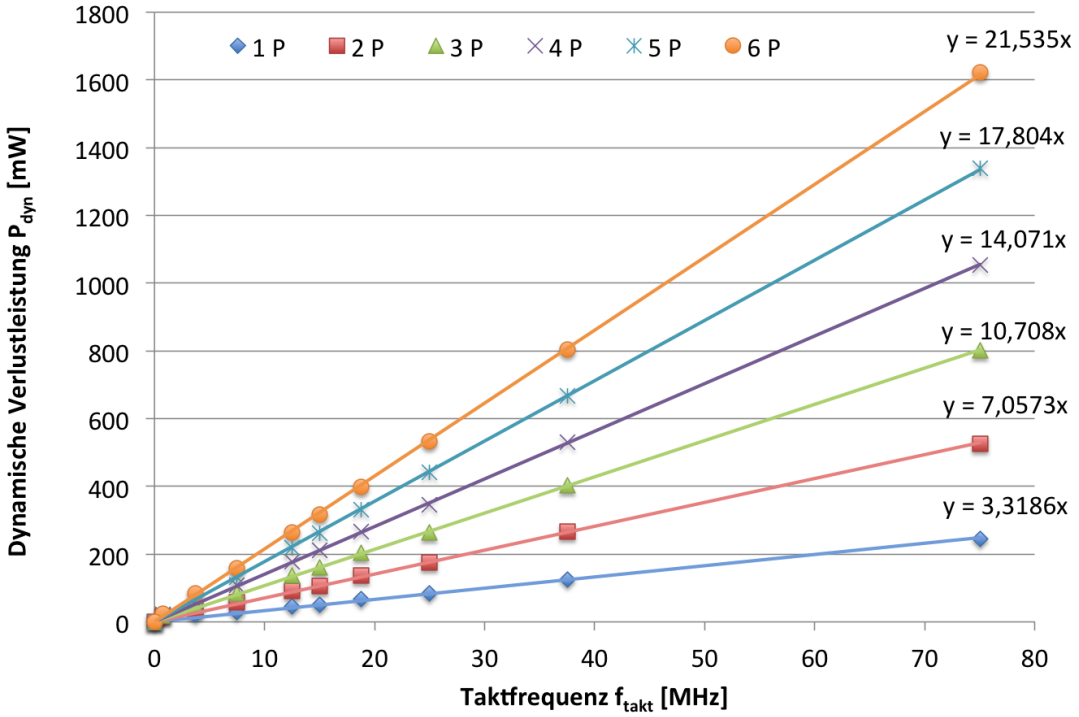


Abbildung B.3: Abhängigkeit der dynamischen Verlustleistung von der Taktfrequenz in Multiprozessorumgebungen verschiedener Größe am Beispiel des Spartan-3

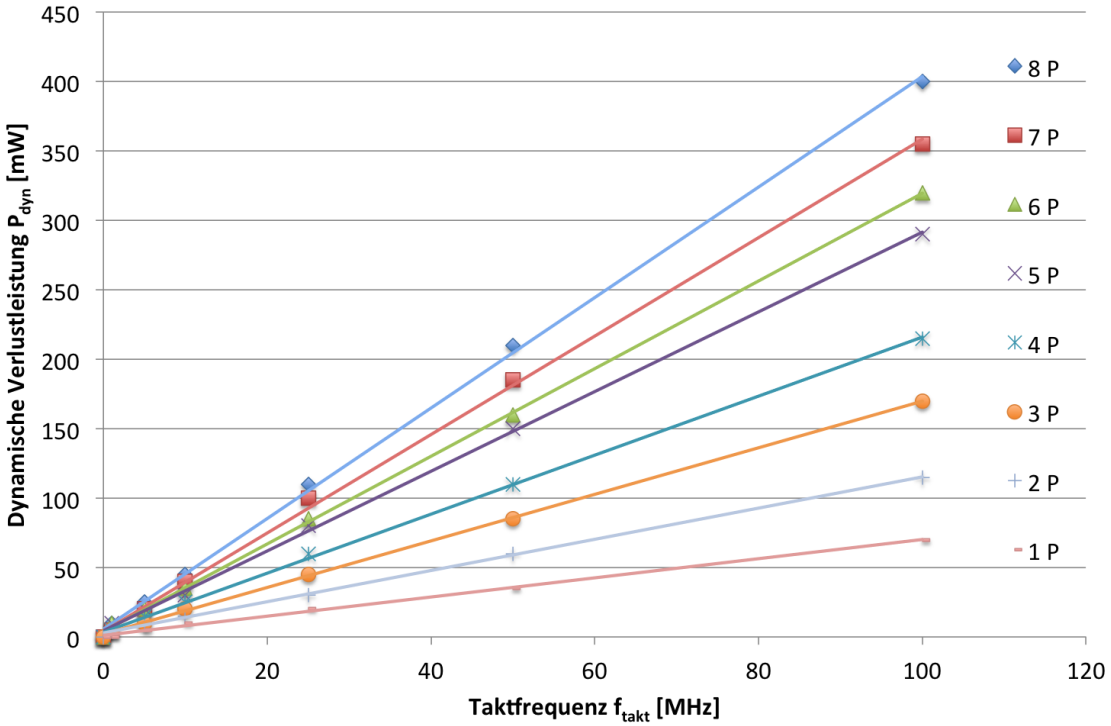


Abbildung B.4: Abhängigkeit der dynamischen Verlustleistung von der Taktfrequenz in Multiprozessorumgebungen verschiedener Größe am Beispiel des Virtex-5

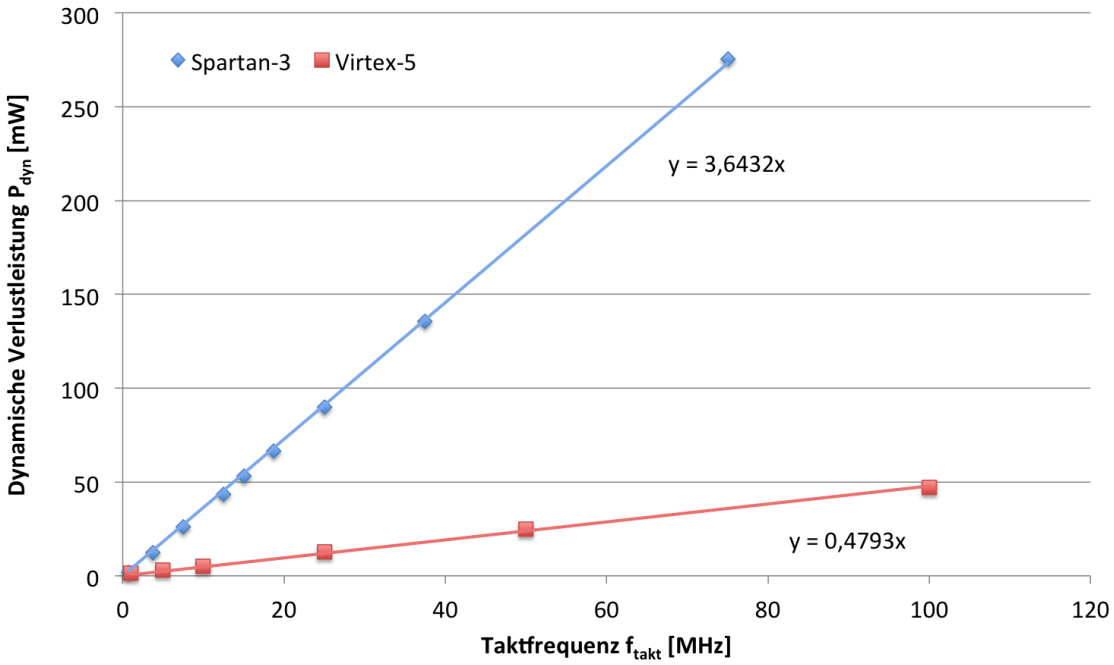


Abbildung B.5: Dynamische Verlustleistung eines einzelnen Softprozessors in Abhängigkeit von der Taktfrequenz am Beispiel des Spartan-3 und des Virtex-5

B.3 Verlustleistung in Abhängigkeit von der Prozessoranzahl

Inhalt. Die Messung soll klären, ob und wie die dynamische Verlustleistung eines Multiprozessorsystems von der Anzahl der Prozessoren in dem Multiprozessorsystem abhängt.

Aufnahme der Messdaten. Für die Untersuchung der Abhängigkeit der dynamischen Verlustleistung P_{dyn} von der Anzahl der Softprozessoren im Multiprozessorsystem werden die Messwerte aus Tabelle B.5, S. 282 (S3-110216 für Spartan-3) und Tabelle B.6, S. 283 (V5-110216 für Virtex-5) herangezogen sowie weiterführende Messungen am Virtex-4 FPGA vorgenommen. Die Größe der Daten- und Befehlsspeicher je Softprozessor beträgt bei der Messreihe V4-110329 (**Tabelle B.8**) ebenfalls je 8 KiB. Hier konnten bis zu 34 Prozessoren auf dem FPGA implementiert werden. Zusätzlich wurde am Virtex-4 die Messreihe V4-110406 (**Tabelle B.9**) mit je 32 KiB Speicher für Daten und Befehle aufgenommen. Aufgrund des größeren Speichers konnten bei dieser Messreihe nur maximal 23 Prozessoren auf dem FPGA implementiert werden. Die beste Genauigkeit bei der Bewertung bieten die Messwerte bei maximaler Taktfrequenz (75 MHz bei Spartan-3 und 100 MHz bei Virtex-4 und Virtex-5), da das Verhältnis von statischer Verlustleistung zu dynamischer Verlustleistung hier besonders günstig ist. Die für die Untersu-

chung verwendeten Messergebnisse aus Tabelle B.5 und Tabelle B.6 sind in **Tabelle B.7** noch einmal aufgeführt. Tabelle B.8 und Tabelle B.9 zeigen die Ergebnisse der neuen Messungen am Virtex-4 FPGA.

Hinweis: Der vom FPGA-Board aufgenommene Strom I_B hängt von der Eingangsspannung U_B ab. Aus diesem Grund muss die Spannungsquelle U_0 während der Messung so eingestellt werden, dass die Eingangsspannung am Board konstant $U_B > 4{,}75\ V$ (Spartan-3), $U_B = 12\ V$ (Virtex-4) bzw. $U_B = 5\ V$ (Virtex-5) beträgt.

Ermittlung der dynamischen Verlustleistung. Die dynamische Verlustleistung am Virtex-4 Board ergibt sich aus der Versorgungsspannung $U_B = 12\ V$ und dem gemessenen Strom I_B, abzüglich der statischen Grundlast I_{Bstat} bei einer Taktfrequenz von $f_{takt} = 0\ MHz$ (Gleichung B.7).

$$P_{dyn} = \frac{U_B}{I_B - I_{Bstat}} \qquad (B.7)$$

Die ermittelten Werte der dynamischen Verlustleistung P_{dyn} in Abhängigkeit von der Anzahl der Prozessoren sind in Tabelle B.8 und Tabelle B.9 aufgelistet.

Tabelle B.7: Datenanalyse aus den Messungen S3-110216 und V5-110216

Anzahl der Prozessoren	**1 P**	**2 P**	**3 P**	**4 P**	**5 P**	**6 P**	**7 P**	**8 P**
FPGA	**Dynamische Verlustleistung P_{dyn} [mW]**							
Spartan-3	245	526	801	1054	1339	1623	n/a	n/a
Virtex-5	70	115	170	215	290	320	355	400

Tabelle B.8: Messdaten aus Messung V4-110329

Anzahl der Prozessoren	**Strom I_B [mA]**	**Strom I_{Bstat} [mA]**	**P_{stat} [mW]**	**P_{dyn} [mW]**
1 P	157,4	145,6	1747	142
2 P	169,6	146,3	1756	280
3 P	179,4	147,5	1770	383
4 P	192,9	149,5	1794	521
5 P	201,2	149,6	1795	619
6 P	211,2	149,8	1798	737
7 P	224,8	150,3	1804	894
8 P	242,2	153,4	1841	1066
9 P	242,8	150,9	1811	1103

Anzahl der Prozessoren	Strom I_B [mA]	Strom I_{Bstat} [mA]	P_{stat} [mW]	P_{dyn} [mW]
10 P	253,5	151,3	1816	1226
11 P	260,4	150,6	1807	1318
12 P	276,8	152,8	1834	1488
13 P	282,2	152,3	1828	1559
14 P	292,2	152,7	1832	1674
15 P	303,4	152,5	1830	1811
16 P	313,6	155,6	1867	1896
17 P	320,7	153,6	1843	2005
18 P	334,5	154,5	1854	2160
19 P	341,1	155,0	1860	2233
20 P	348,3	154,4	1853	2327
21 P	360,8	154,4	1853	2477
22 P	364,7	155,0	1860	2516
23 P	378,4	155,7	1868	2672
24 P	388,2	154,6	1855	2803
25 P	395	154	1848	2892
26 P	408	153	1836	3060
27 P	415	155	1860	3120
28 P	424	155	1860	3228
29 P	438	154	1848	3408
30 P	446	154	1848	3504
31 P	448	151	1812	3564
32 P	461	154	1848	3684
33 P	478	153	1836	3900
34 P	477	153	1836	3888

Tabelle B.9: Messdaten aus Messung V4-110406

Anzahl der Prozessoren	Strom I_B [mA]	Strom I_{Bstat} [mA]	P_{stat} [mW]	P_{dyn} [mW]
1 P	160,0	144,0	1728	192
2 P	177,4	145,5	1746	383
3 P	191,1	146,2	1754	539
4 P	208,1	148,7	1784	713
5 P	221,9	148,7	1784	878
6 P	242,4	150,2	1806	1103
7 P	260,8	152,6	1831	1298
8 P	271,5	151,5	1818	1440
9 P	278,1	151,4	1817	1520

Anzahl der Prozessoren	Strom I_B [mA]	Strom I_{Bstat} [mA]	P_{stat} [mW]	P_{dyn} [mW]
10 P	301,9	154,8	1858	1765
11 P	309,8	152,6	1831	1886
12 P	324,6	153,9	1847	2048
13 P	343,4	155,2	1862	2258
14 P	355,4	154,9	1859	2406
15 P	359,8	153,3	1840	2478
16 P	381,5	155,3	1864	2714
17 P	391,2	155,0	1860	2834
18 P	404	154	1848	3000
19 P	420	154	1848	3192
20 P	436	155	1860	3372
21 P	440	152	1824	3456
22 P	460	157	1884	3636
23 P	473	156	1872	3804

Abhängigkeit von der Anzahl der Prozessoren. Die Werte aus Tabelle B.7 und Tabelle B.8 sind in der **Abbildung B.6** und der **Abbildung B.7** grafisch dargestellt. Deutlich sichtbar ist in allen Messreihen ein linearer Verlauf der Abhängigkeit der dynamischen Verlustleistung P_{dyn} von der Anzahl der Softprozessoren im Multiprozessorsystem, der den eingezeichneten Regressionsgeraden folgt. Aus der Steigung der Regressionsgeraden in Abbildung B.6 ergibt sich beim Spartan-3 FPGA für jeden Softprozessor eine dynamische Verlustleistung $P_{dyn} \approx 268\,mW$ bei einer Taktfrequenz f_{takt} = 75 MHz und beim Virtex-5 FPGA ergibt sich für jeden Softprozessor eine dynamische Verlustleistung $P_{dyn} \approx 53\,mW$ bei einer Taktfrequenz $f_{takt} = 100\,MHz$.

Aus der Steigung der Regressionsgeraden in Abbildung B.7 ergibt sich beim Virtex-4 FPGA für jeden Softprozessor eine dynamische Verlustleistung $P_{dyn} \approx 53\,mW$ bei einer Taktfrequenz f_{takt}= *100 MHz*. Mit der in Abbildung B.3 und Abbildung B.4, S. 285 nachgewiesenen linearen Abhängigkeit der dynamischen Verlustleistung P_{dyn} von der Taktfrequenz f_{takt} ergibt sich für die Softprozessoren im Spartan-3 FPGA eine vergleichbare dynamische Verlustleistung $P_{dyn}(f_{takt}) \approx 3{,}6\,mW/MHz$ (Gleichung B.8) und im Virtex-5 FPGA eine deutlich geringere dynamische Verlustleistung $P_{dyn}(f_{takt}) \approx 0{,}53\,mW/MHz$ (Gleichung B.10). Zwischen diesen beiden FPGAs liegt der Virtex-4 FPGA mit einer dynamischen Verlustleistung $P_{dyn}(f_{takt}) \approx 1{,}2\,mW/MHz$ (Gleichung B.9).

Spartan-3 $$P_{dyn}\left(f_{takt}\right) \approx \frac{268\,mW}{75\,MHz} \approx 3{,}6\,\frac{mW}{MHz} \qquad (B.8)$$

Virtex-4 $$P_{dyn}\left(f_{takt}\right) \approx \frac{117\,mW}{100\,MHz} \approx 1{,}2\,\frac{mW}{MHz} \qquad (B.9)$$

Virtex-5 $$P_{dyn}\left(f_{takt}\right) \approx \frac{53\,mW}{100\,MHz} \approx 0{,}53\,\frac{mW}{MHz} \qquad (B.10)$$

Die **Abbildung B.8** zeigt den Verlauf der Anteile von statischer und dynamischer Verlustleistung an der Gesamtverlustleistung in Abhängigkeit von der Anzahl der Prozessoren. Während die statische Verlustleistung über die gesamte Messreihe unverändert bleibt, steigt der Anteil der dynamischen Verlustleistung mit jedem zusätzlichen Softprozessor. Grundsätzlich kann man daraus ablesen, dass mit der Größe des Multiprozessorsystems auch das Einsparpotential steigt.

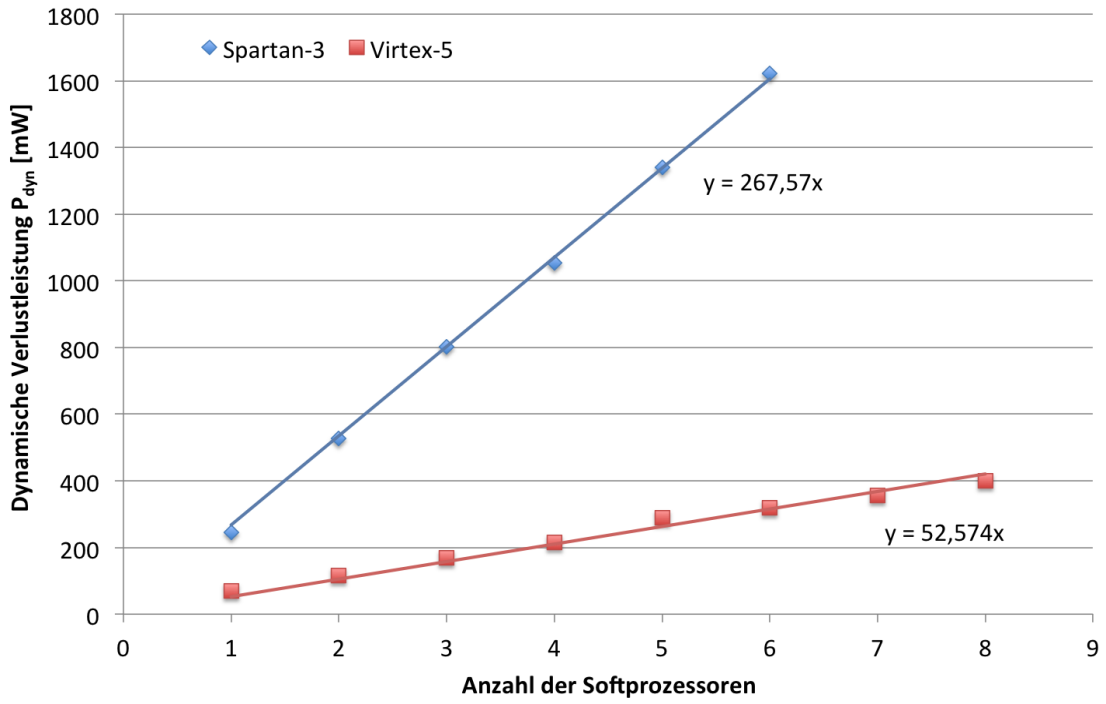


Abbildung B.6: Abhängigkeit der dynamischen Verlustleistung von der Anzahl der Softprozessoren im Multiprozessorsystem am Beispiel des Spartan-3 und Virtex-5

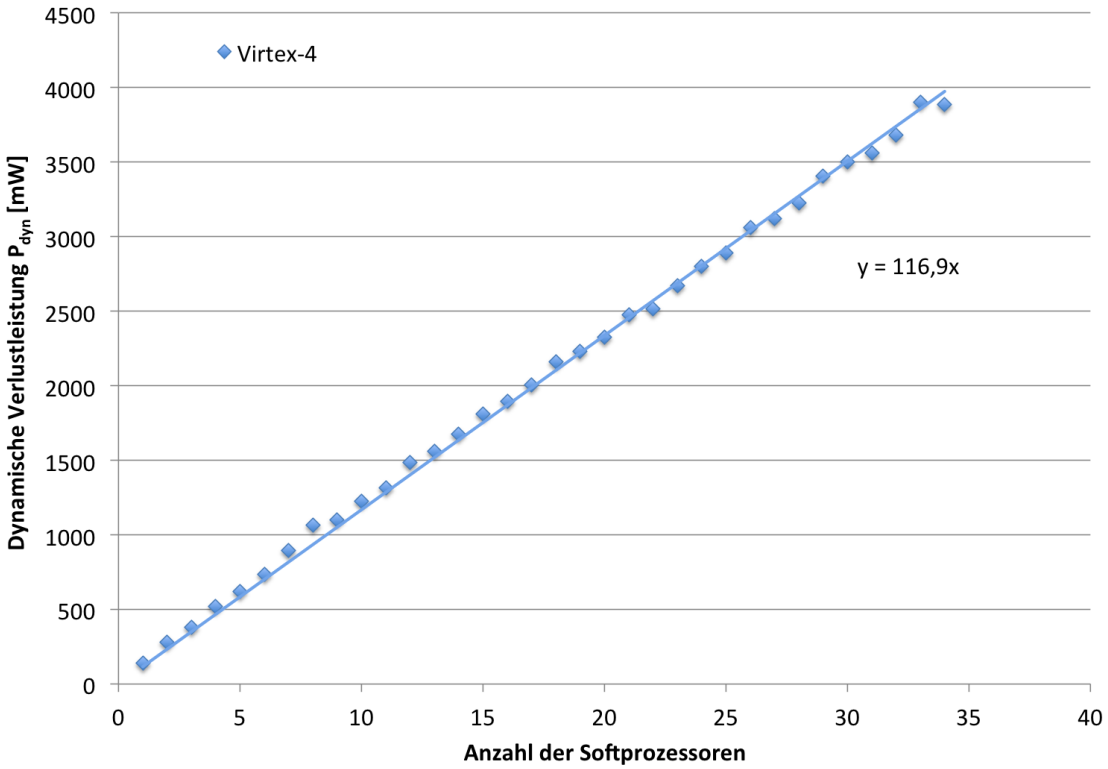


Abbildung B.7: Abhängigkeit der dynamischen Verlustleistung von der Anzahl der Softprozessoren im Multiprozessorsystem am Beispiel des Virtex-4

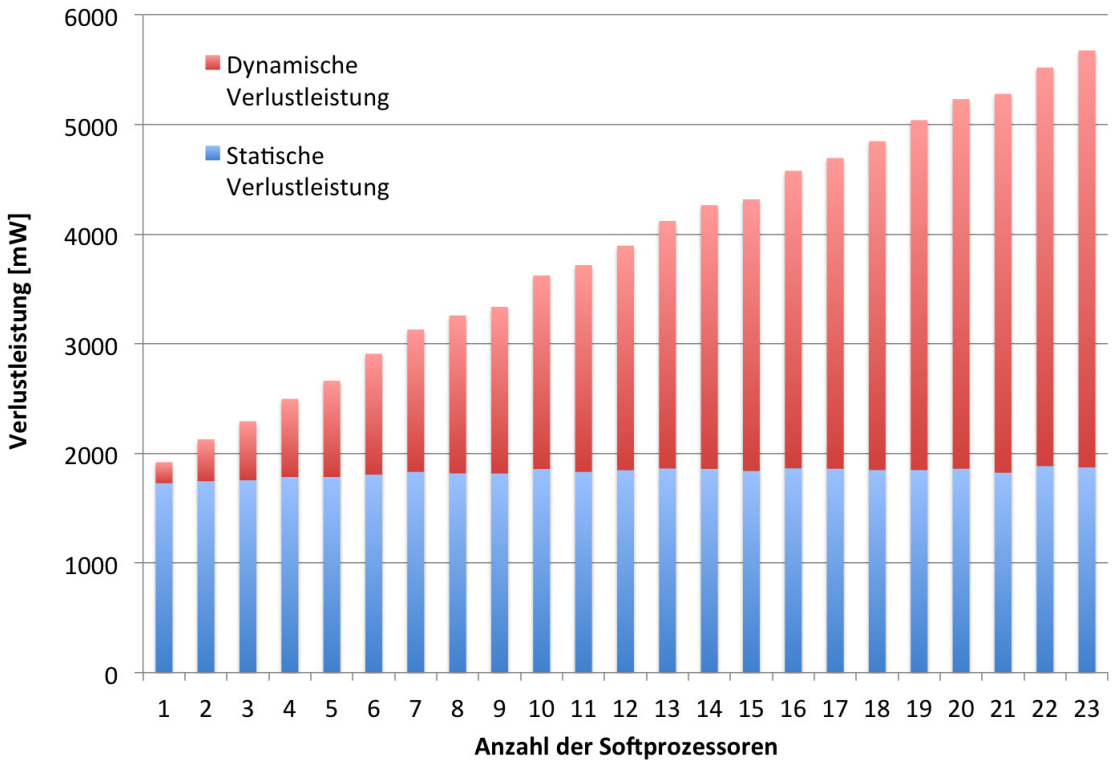


Abbildung B.8: Abhängigkeit der statischen und der dynamischen Verlustleistung von der Anzahl der Softprozessoren im Multiprozessorsystem am Beispiel des Virtex-4

B.4 Verlustleistung in Abhängigkeit von der Speichergröße

Inhalt. Diese Messung soll klären, ob und wie die dynamische Verlustleistung eines Multiprozessorsystems von der Größe der Lokalspeicher der Prozessoren abhängt.

Aufnahme der Messdaten. Die nachfolgenden Messdaten wurden am Virtex-4 Board aufgenommen. Dieses Board besitzt zwei getrennte Stromversorgungen für FPGA und Peripherie, wodurch die Messergebnisse am FPGA relativ präzise ausfallen. Es wurde jeweils eine Umgebung aus 1 bis 5 Softprozessoren implementiert. Die Taktfrequenz der Prozessoren beträgt fest 100 MHz. Für die Ermittlung der statischen Verlustleistung kann der Takt über einen DIP-Schalter abgeschaltet werden, so dass die Prozessoren nicht mehr getaktet werden. Die Speichergröße für Befehls- und Datenspeicher je Softprozessor variiert in den Messungen von je 16 KiB bis 128 KiB. Daraus ergibt sich eine gesamte Speichergröße mit 32 KiB bis 256 KiB je Softprozessor. Die Stromaufnahme wurde am Anschluss P3 (hard drive connector) gemessen. Das Board wird von extern mit einer Spannung von 12 V versorgt, die intern in 1,8 V für die SDRAM und in 1,2 V für den Virtex-4 FPGA gewandelt wird [Pl06]. Die in den Messungen ermittelten Messdaten sind in der **Tabelle B.10** dokumentiert.

Hinweis: Der vom FPGA-Board aufgenommene Strom I_B hängt von der Eingangsspannung U_B ab. Aus diesem Grund muss die Spannungsquelle U_0 während den Messungen so eingestellt werden, dass die Eingangsspannung konstant $U_B = 12\ V$ beträgt.

Ermittlung der dynamischen Verlustleistung. Die dynamische Verlustleistung ergibt sich aus der Versorgungsspannung $U_B = 12\ V$ und dem gemessenen Strom I_B, abzüglich der statischen Grundlast I_{Bstat} bei einer Taktfrequenz von $f_{takt} = 0\ MHz$ (Gleichung B.11).

$$P_{dyn} = \frac{U_B}{I_B - I_{Bstat}} \qquad (B.11)$$

Tabelle B.10: Messdaten aus Messung V4-110314

Anzahl der Prozessoren	Speichergröße je Prozessor	Strom I_B [mA]	Strom I_{Bstat} [mA]
1 P	32 KiB	158,0	143,7
	64 KiB	159,9	143,1
	128 KiB	166,2	143,8
	256 KiB	176,0	143,2
2 P	32 KiB	168,8	143,9
	64 KiB	176,9	145,9
	128 KiB	186,8	145,3
	256 KiB	211,0	147,8
3 P	32 KiB	190,8	149,8
	64 KiB	190,2	145,7
	128 KiB	207,2	146,9
	256 KiB	248,1	152,0
4 P	32 KiB	200,9	149,0
	64 KiB	211,9	150,3
	128 KiB	229,1	147,4
	256 KiB	284,6	155,1
5 P	32 KiB	207,3	147,7
	64 KiB	223,2	149,6
	128 KiB	247,5	148,9
	256 KiB	309,3	153,2

Tabelle B.11: Datenanalyse aus Messung V4-110314

Speicher je Prozessor	P_{dyn} [mW] (1 P)	P_{dyn} [mW] (2 P)	P_{dyn} [mW] (3 P)	P_{dyn} [mW] (4 P)	P_{dyn} [mW] (5 P)
32 kB	171,6	298,8	492,0	622,8	715,2
64 kB	201,6	372,0	534,0	739,2	883,2
128 kB	268,8	498,0	723,6	980,4	1183,2
256 kB	393,6	758,4	1153,2	1554,0	1873,2

Tabelle B.12: Erweiterte Datenanalyse aus Messung V4-110314

Dyn. Verlustleistung P_{dyn}	1 P	2 P	3 P	4 P	5 P
Steigung [mW/KiB]	0,995	2,038	3,038	4,178	5,157
Offset [mW]	139,5	237,2	361,2	472,8	544,9
Steigung pro MB [mW/KiB]	n/a	1,043	1,021	1,061	1,041
Offset pro MB [mW]	n/a	97,7	110,8	111,1	101,3

Die ermittelten Werte der dynamischen Verlustleistung P_{dyn} in Abhängigkeit von der Größe der Lokalspeicher sind in der **Tabelle B.11** aufgelistet.

Abhängigkeit von der Größe der Lokalspeicher. Die ermittelten Werte der dynamischen Verlustleistung P_{dyn} (Tabelle B.11) sind in der **Abbildung B.9** grafisch dargestellt. Für jede Multiprozessorumgebung ist je eine Datenreihe mit unterschiedlich großen Lokalspeichern aufgetragen. Sofort auffällig ist in den Messpunkten der lineare Zusammenhang zwischen der dynamischen Verlustleistung P_{dyn} und der Größe der Lokalspeicher. Deshalb wurde für jede Datenreihe eine Regressionsgerade eingetragen, deren Steigung und Schnittpunkt mit der vertikalen Achse ebenfalls abgebildet sind. Auffällig ist an dieser Stelle der Zusammenhang zwischen den Datenreihen, sowohl in der Steigung als auch im Offset.

Die **Tabelle B.12** zeigt eine weiterführende Analyse der Daten aus der Messung V4-110314, die sich mit Hilfe der Regressionsgeraden aus der Abbildung B.9 ermitteln lassen. Hierzu werden die Steigungen und Offsets der Regressionsgeraden verwendet. Die Steigung der Geraden steht für den Einfluss der Größe der Lokalspeicher auf die dynamische Verlustleistung P_{dyn}. Der Schnittpunkt der Geraden mit der vertikalen Achse (Offset) zeigt den Teil der dynamischen Verlustleistung, den die Logikgatter des Softprozessors verbrauchen. Um ein relativ genaues Bild zu bekommen, werden nur die Abstände zwischen den Geraden verwendet, da hier der Abstand genau einen Softprozessor inklusive seiner Lokalspeicher beträgt. Für jede Multiprozessorumgebung werden die Werte aus dem Einzelprozessorsystem abgezogen und die Differenz durch die Anzahl der Prozessoren geteilt. Wie sich aus den Ergebnissen in der Tabelle B.12 ablesen lässt, kann bei dem verwendeten Virtex-4 FPGA für jedes KiB Speicher etwa 1 mW als dynamische Verlustleistung eingeplant werden. Für jeden Softprozessor kommen noch einmal etwa 100 mW hinzu. Mit der bereits bekannten linearen Abhängigkeit der dynamischen Verlustleistung von der Taktfrequenz und den Werten aus der Tabelle B.12 ergeben sich für den verwendeten Virtex-4 FPGA die in der Gleichung B.12 beschriebenden Zusammenhänge.

$$P_{dyn}(MB) = 1\frac{mW}{MHz} + 0{,}01\frac{mW}{MHz \cdot KiB} \qquad (B.12)$$

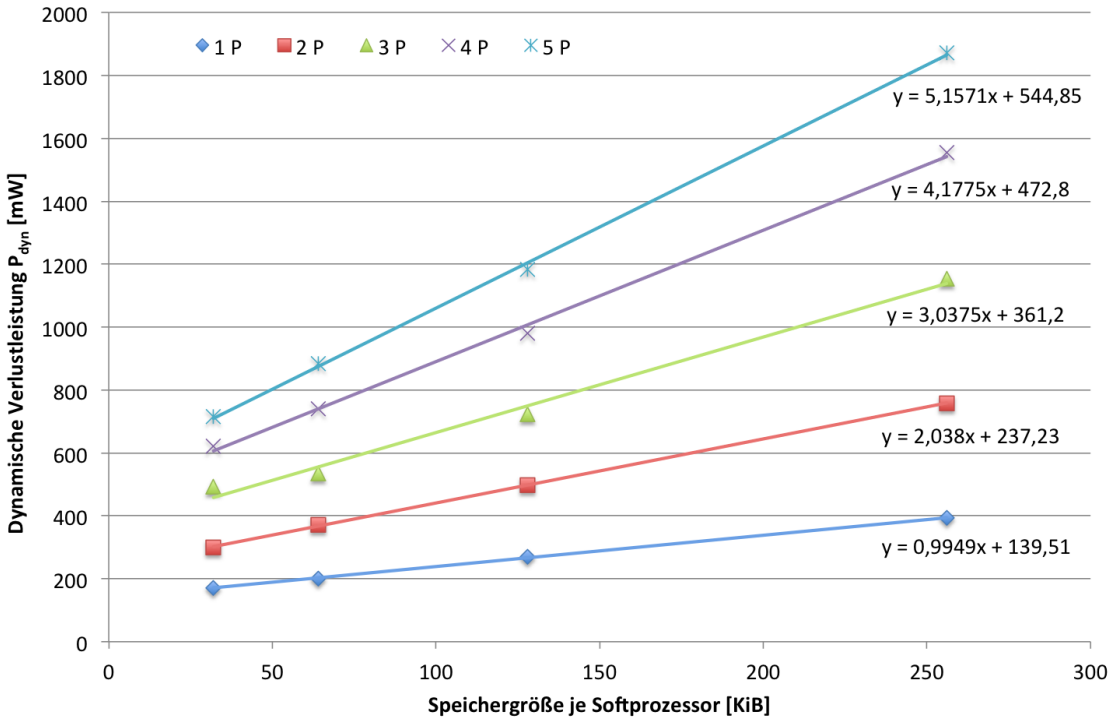


Abbildung B.9: Abhängigkeit der dynamischen Verlustleistung von der Speichergröße bei einer Taktfrequenz von 100 MHz am Beispiel des Virtex-4

B.5 Verlustleistung in Abhängigkeit vom Verbindungsnetzwerk

Inhalt. Die Messung soll klären, ob und wie die dynamische Verlustleistung eines Multiprozessorsystems von der Übertragungsgeschwindigkeit des Verbindungsnetzwerkes abhängt.

Aufnahme der Messdaten. Die nachfolgenden Messdaten wurden am Virtex-4 Board aufgenommen. Dieses Board besitzt zwei getrennte Stromversorgungen für FPGA und Peripherie, wodurch die Messergebnisse am FPGA relativ präzise ausfallen. Es wurde jeweils ein Multiprozessorsystem mit 8 Softprozessoren implementiert. Die Taktfrequenz der Prozessoren beträgt fest 100 MHz. Die Interprozessorkommunikation erfolgt durch ein dynamischesVerbindungsnetzwerk. Die Kopplung zwischen dem Verbindungsnetzwerk und den Prozessoren erfolgt über den proprietären FSL-Bus von Xilinx. Die zeitliche Entkopplung der Verbindungen erfolgt durch 512 Byte große FIFO-Puffer auf beiden Seiten des Netzwerks. Dadurch wird die asynchrone Kommunikation zwischen den unterschiedlich getakteten Prozessoren ermöglicht. Das Verbindungsnetzwerk besteht aus einem einfachen nicht-blockierungsfreien logN-Netz mit einer Baseline-Topologie der

Größe 8x8. Das Verbindungsnetzwerk wird in den Messungen mit unterschiedlichen festen Taktraten betrieben. Für die Ermittlung der statischen Verlustleistung kann der Takt über einen DIP-Schalter abgeschaltet werden, so dass das Verbindungsnetzwerk nicht mehr getaktet wird. Die Stromaufnahme wurde am Anschluss P3 (*hard drive connector*) gemessen. Das Board wird von extern mit einer Spannung von 12 V versorgt, die intern in 1,8 V für die SDRAM und in 1,2 V für den Virtex-4 FPGA gewandelt wird [Pl06]. Die in den Messungen ermittelten Messdaten sind in der **Tabelle B.13** dokumentiert.

Tabelle B.13: Messdaten aus Messung V4-110324

f_{netz} [MHz]	Strom I_B [mA]	Strom I_{Bstat} [mA]	Dyn. Verlustleistung P_{dyn} [mW]
12,5	179,9	179,1	9,6
25	181,4	179,7	20,4
37,5	178,2	175,6	31,2
50	183,2	179,7	42,0
62,5	184,7	180,3	52,8
75	182,9	177,6	63,6
87,5	188,4	182,2	74,4
100	195,6	188,5	85,2

Hinweis: Der vom FPGA-Board aufgenommene Strom I_B hängt von der Eingangsspannung U_B ab. Aus diesem Grund muss die Spannungsquelle U_0 während den Messungen so eingestellt werden, dass die Eingangsspannung konstant $U_B = 12\ V$ beträgt.

Ermittlung der dynamischen Verlustleistung. Die dynamische Verlustleistung ergibt sich aus der Versorgungsspannung $U_B = 12\ V$ und dem gemessenen Strom I_B, abzüglich der statischen Grundlast I_{Bstat} bei einer Taktfrequenz von $f_{netz} = 0\ MHz$ (Gleichung B.13).

$$P_{dyn} = \frac{U_B}{I_B - I_{Bstat}} \qquad (B.13)$$

Die ermittelten Werte der dynamischen Verlustleistung P_{dyn} sind in Abhängigkeit von der Geschwindigkeit des Verbindungsnetzwerks in der Tabelle B.12 aufgelistet.

Abhängigkeit vom Verbindungsnetzwerk. Die ermittelten Messdaten aus der Tabelle B.12 sind in der **Abbildung B.10** grafisch dargestellt. Sofort auffällig ist ein linearer Zusammenhang zwischen der Taktfrequenz f_{netz} und der dynamischen Verlustleistung P_{dyn}. In die Messpunkte wurde eine Regressionsgerade gelegt, deren Steigung ebenfalls dargestellt ist. Anhand der Steigung der Geraden lässt sich die Abhängigkeit beider Größen ablesen (Gleichung B.14).

$$P_{dyn}(f_{netz}) \approx 0,85 \frac{mW}{MHz} \qquad (B.14)$$

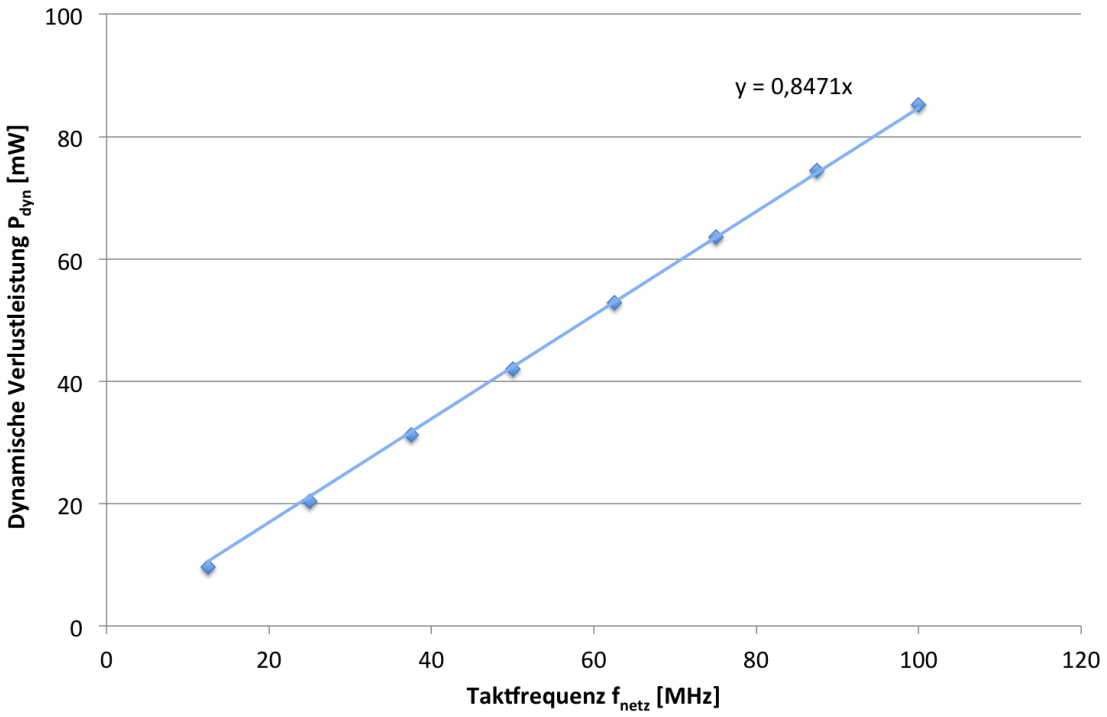


Abbildung B.10: Abhängigkeit der dynamischen Verlustleistung des Verbindungsnetzwerks der Größe 8x8 von dessen Taktfrequenz am Beispiel des Virtex-4

B.6 Verlustleistung in Abhängigkeit vom Prozessordesign

Inhalt. Die Messung soll zeigen, wie die einzelnen Komponenten im Design eines Softprozessors die Verlustleistung beeinflussen und welche Auswirkungen ein Kompromiss zugunsten einer verringerten Rechenleistung im Hinblick auf den Energieverbrauch besitzt. Zusätzlich dazu soll geklärt werden, welche leistungssteigernden Komponenten den Energieverbrauch des Systems in besonderem Maße belasten.

Aufnahme der Messdaten. Die Messungen wurden am Spartan-3 Board durchgeführt. Hierfür wurde ein MicroBlaze Softprozessor in verschiedenen Ausbaustufen synthetisiert und der jeweilige Leistungsbedarf ermittelt. Zunächst wurde nur die Basisfunktionalität implementiert, später kamen sukzessiv weitere optionale Komponenten hinzu[1].

Ermittlung der Verlustleistung. Bei der Verlustleistung soll bei dieser Messung die Gesamtverlustleistung im Vordergrund stehen. Aus diesem Grund wird der gemessene Strom I_B jeder Ausbaustufe des Prozessors mit der Versorgungsspannung $U_B = 5\ V$ zur Leistung P_{ges} multipliziert. Da jeweils eine weitere Komponente im Design hinzugefügt wurde, ergibt sich die Verlustleistung für jede Komponente aus der Differenz zwischen dem aktuellen und dem vorhergehenden Messwert. Die daraus resultierenden Ergebnisdaten sind zusammen mit den Messdaten in der **Tabelle B.14** dokumentiert.

Tabelle B.14: Messdaten aus Messung S3-111205

Ausbaustufe	**Strom IB [mA]**	**Verlustleistung P_{ges} [mW]**	**Verlustleistung P [mW]**
Grundlast (ohne MB)	214,5	1073	-
Basic Functionality	250,9	1255	182
+ 5-stage Pipeline	272,3	1362	107
+ Barrel Shifter	274,9	1375	13
+ 32-Bit Multiplier	277,1	1386	11
+ Integer Divider	280,1	1401	15
+ Floating Point Unit	303,9	1520	119

Zusätzlicher Energiebedarf. Wie sich unschwer aus der Tabelle B.14 ersehen lässt, erhöht sich der Energiebedarf des Rechensystems mit jeder zusätzlichen Komponente, um welche der Softprozessor erweitert wird. Auf diese Weise erhöht sich die Verlustleistung des Softprozessors von 182 mW auf insgesamt 447 mW. Der Energiebedarf eines MicroBlazes der vollen Ausbaustufe entspricht damit etwa dem 2,5-fachen des Energiebedarfs eines MicroBlazes mit Basisfunktionalität.

[1] vgl. Abbildung 4.13: Anteiliger Flächenbedarf im Softprozessor, S. 67

Lebenslauf

Persönliche Daten:

Aust, Stefan
geboren am 30. April 1974 in Eisenhüttenstadt
verheiratet, 3 Kinder
deutsch

Berufserfahrung:

seit 10/2012	Volkswagen AG, Wolfsburg Technische Entwicklung Elektrik-/Elektronikentwicklung
03/2009 - 09/2012	TU Clausthal, Clausthal-Zellerfeld Institut für Informatik Abteilung Technische Informatik und Rechnersysteme wissenschaftlicher Mitarbeiter
03/2003 - 12/2008	OPTIMA GROUP pharma GmbH, Schwäbisch Hall Softwareingenieur im pharmazeutischen Sondermaschinenbau
09/2001 - 02/2003	EPISOL GmbH, Crailsheim Ingenieurdienstleister
09/1999 - 03/2000	DaimlerChrysler Aerospace AG (heute: EADS), Ulm Verteidigung und Zivile Systeme, Entwicklung Bordsysteme Praxissemester
02/1994 - 09/1997	Unitechnik Automatisierungs GmbH, Eisenhüttenstadt Elektromonteur u.a. für Stahl- und Walzwerke, Logistik-Center, Gebäudeautomation

Ausbildung:

seit 09/2009	TU Clausthal, Clausthal-Zellerfeld Promotionsstudium
10/1997 - 08/2001	Fachhochschule Lausitz, Senftenberg Studium der Elektrotechnik Fachrichtung Automatisierungs- und Mikrosystemtechnik Abschluss: Diplom-Ingenieur (FH), Note 1,2 Diplomstudienpreis des VDI Berlin-Brandenburg
09/1994 - 08/1995	Fachoberschule, Oberstufenzentrum Eisenhüttenstadt Abschluss: Fachhochschulreife
09/1990 - 01/1994	EKO Stahl GmbH, Eisenhüttenstadt Ausbildung zum Energieelektroniker

Sonstige:

01/2009 - 02/2009	Elternzeit
01/1996 - 10/1996	Wehrdienst bei der Bundeswehr

Publikationen

Aust, S.; Richter, H.: *Parallel Computer Technology - A Solution for Automobiles? How car engineers can learn from parallel computing.* 3rd International Conference on Advanced Engineering Computing and Applications in Sciences; ADVCOMP 2010, Sliema, Oktober 2009, S. 148-152.

Aust, S.; Richter, H.: *Space Division of Processing Power for Feed Forward and Feed Back Control in Complex Production and Packaging Machinery.* World Automation Congress; WAC 2010, Kobe, September 2010, S. 1-6.

Aust, S.; Richter, H.: *Ein Echtzeitparallelrechner zur Rezentralisierung von Steuergeräten im Automobil.* erschienen in: Tschöke, H. (Hrsg.), Krahl, J. (Hrsg.), Munack, A. (Hrsg.): *Innovative Automobiltechnik II.* Expert Verlag, 2010, S. 70-88.

Aust, S.; Richter, H.: *Real-time Processor Interconnection Network for FPGA-based Multiprocessor System-on-Chip (MPSoC).* 4th International Conference on Advanced Engineering Computing and Applications in Sciences; ADVCOMP 2010, Florenz, Oktober 2010, S. 47-52.

Aust, S.; Richter, H.: *Skalierbare Rechensysteme für Echtzeitanwendungen.* erschienen in: Halang, W. A. (Hrsg.): *Herausforderungen durch Echtzeitbetrieb.* Springer Verlag, 2011, S. 111-120.

Best Paper Award

Aust, S.; Richter, H.: *Energy-Aware MPSoC with Space-Sharing for Real-Time Applications.* 5th International Conference on Advanced Engineering Computing and Applications in Sciences; ADVCOMP 2011, Lissabon, November 2011, S. 54-59.

Aust, S.; Richter, H.: *Energy-aware MPSoC for Real-time Applications with Space-Sharing,* Adaptive and Selective Clocking and Software-first Design. International Journal on Advances in Software, Vol. 5, Nr. 3 & 4, 2012, S. 368-377.

Vorträge

„Ein Echtzeitparallelrechner zur Rezentralisierung von Steuergeräten im Automobil“
Wissenschaftssymposium Automobiltechnik 2010, Coburg, 14.-15. Oktober 2010

„Skalierbare Rechensysteme für Echtzeitanwendungen“
Gastvortrag, Universität Osnabrück, 13. April 2011

„Scalable Data Processing Systems for Real-Time Applications“
5th Symtavision NewsConference, Braunschweig, 05.-06 Oktober 2011

„Skalierbare Rechensysteme für Echtzeitanwendungen“
Workshop Echtzeit 2011 der GI, Boppard, 03.-04. November 2011